Computational Motion Planning

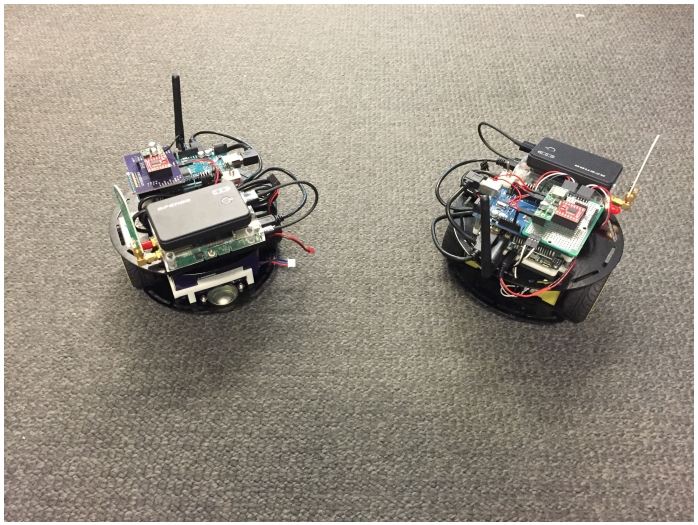
Prof. C.J. Taylor

SECTION 1.1 - INTRO

The Motion Planning Problem

- A special case of the more general planning problem
- The goal is to develop techniques that would allow a robot or robots to automatically decide how to move from one position or configuration to another.
  - Specifically concerned with planning motions – get robot from place A to B

Motion Planning for Robotics



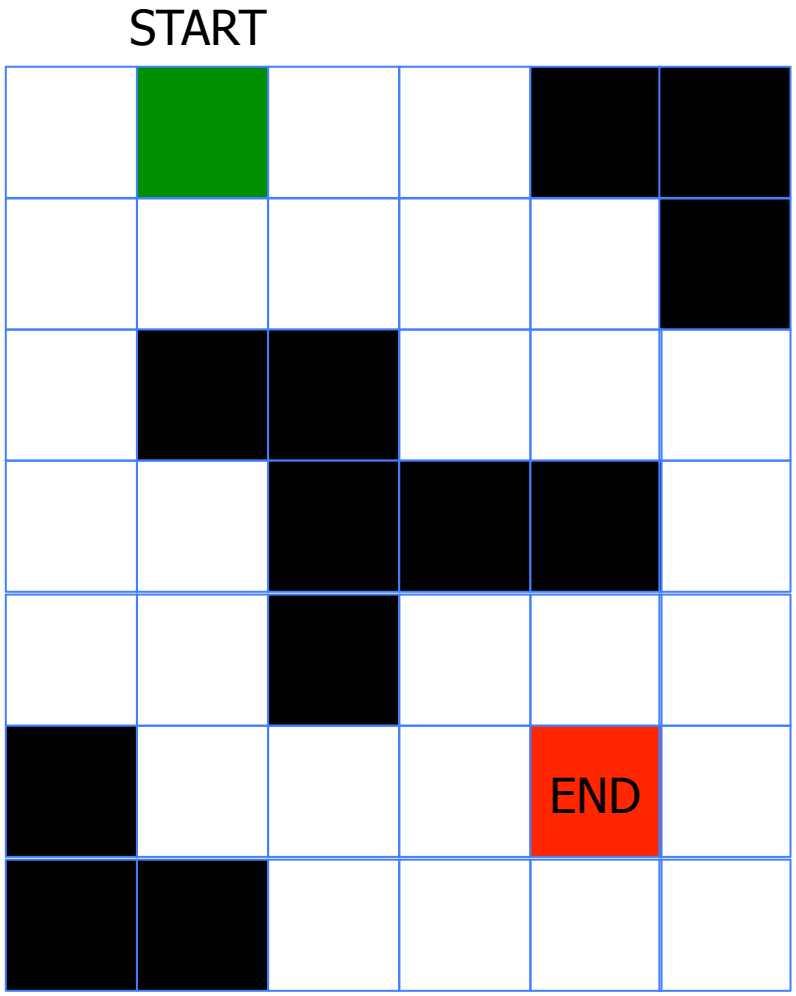
Penn

An Example – the PacMan problem

- How does the computer guide the ghosts back to their lair when they are eaten?

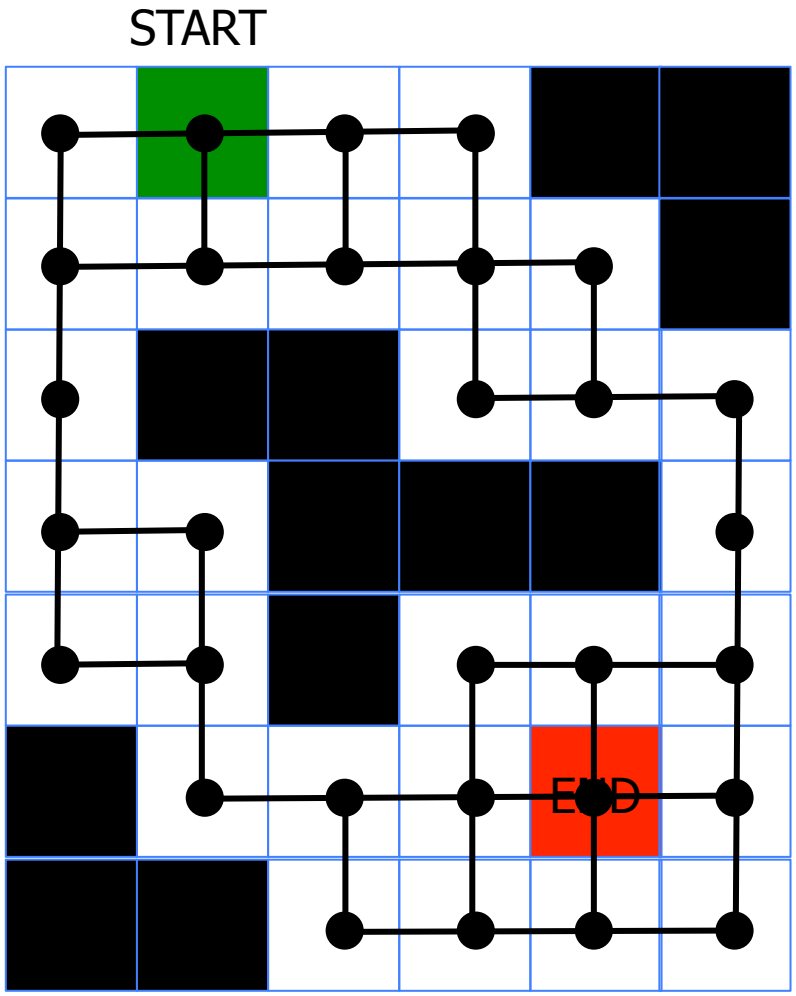
Planning on a grid

- In this example the robot can move between adjacent cells on the grid
- The dark squares indicate obstacles that the robot cannot traverse.



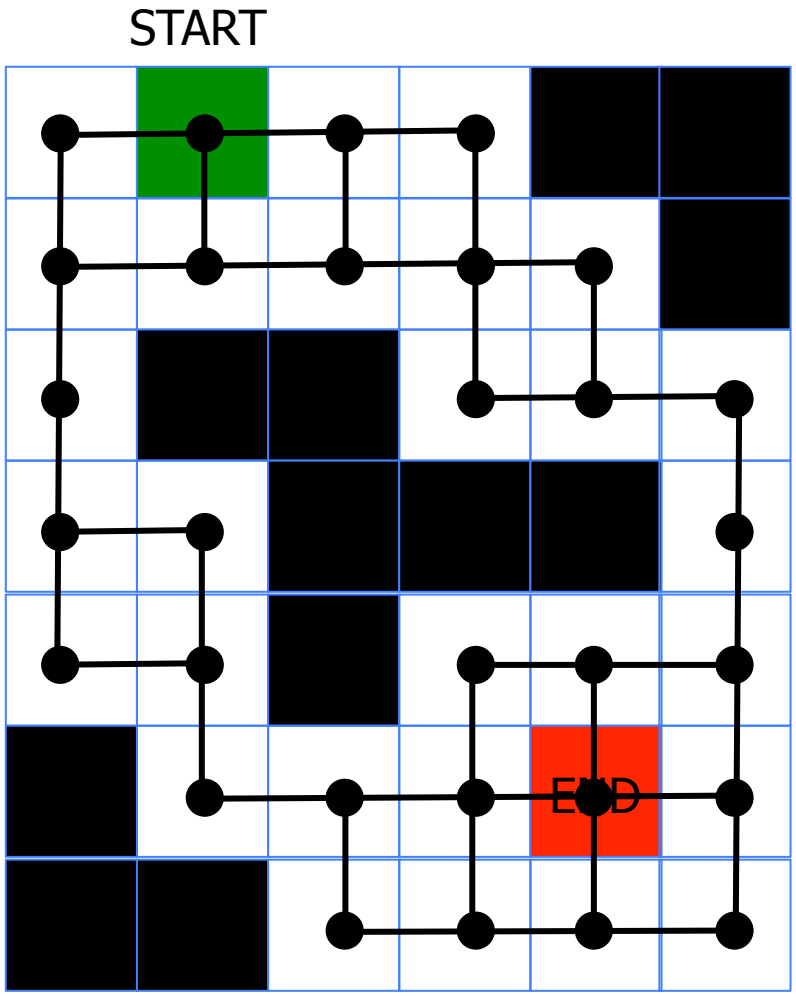
Graph Structure

- We can think of the unoccupied cells as nodes and draw edges between adjacent cells as shown here.
- This set of nodes and edges constitutes a graph.

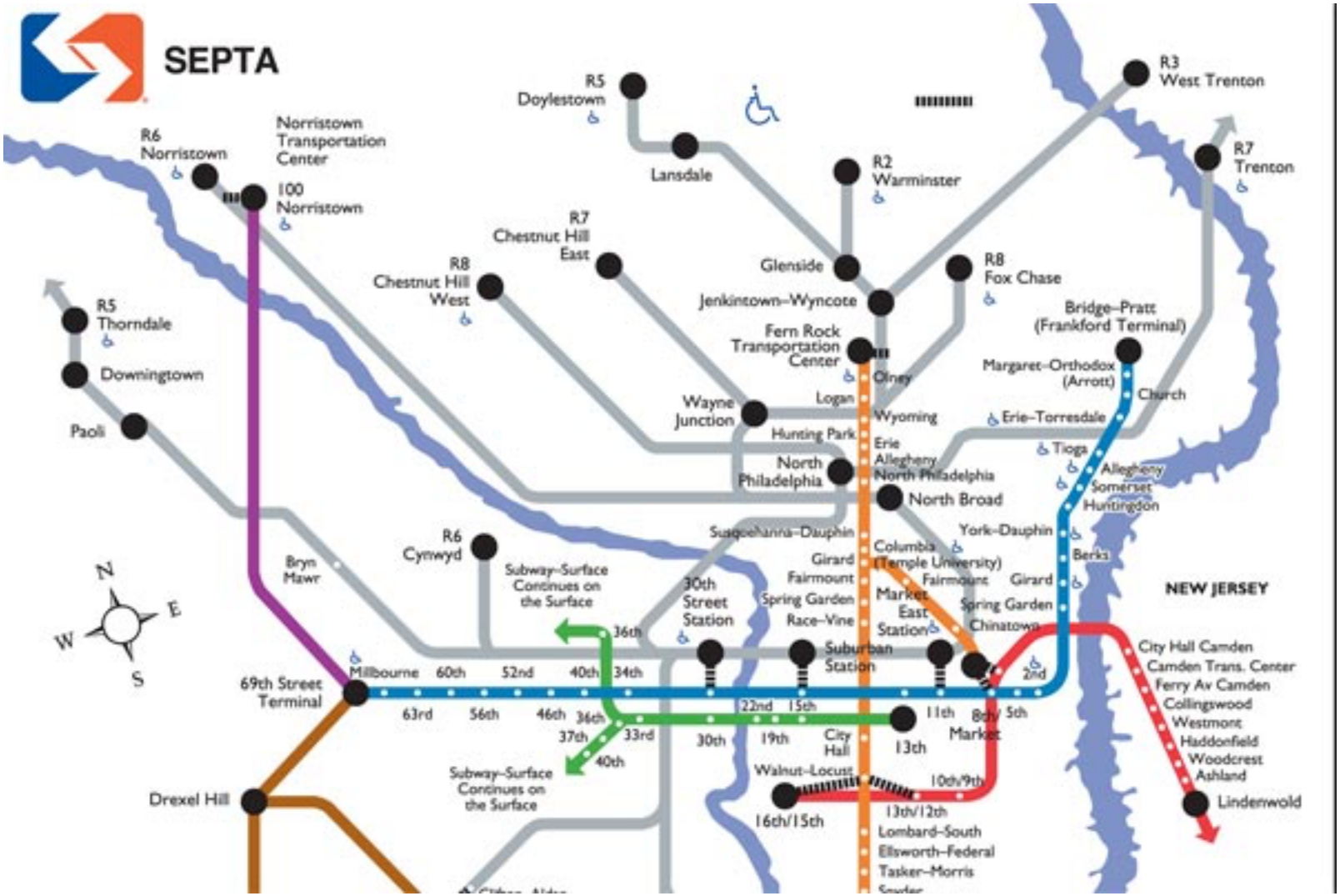


Graph Structure

- A graph, G, consists of a set of vertices, V, and a set of Edges, E, that link pairs of vertices.
- The edges are often annotated with numerical values to indicate relevant quantities like distances or costs.

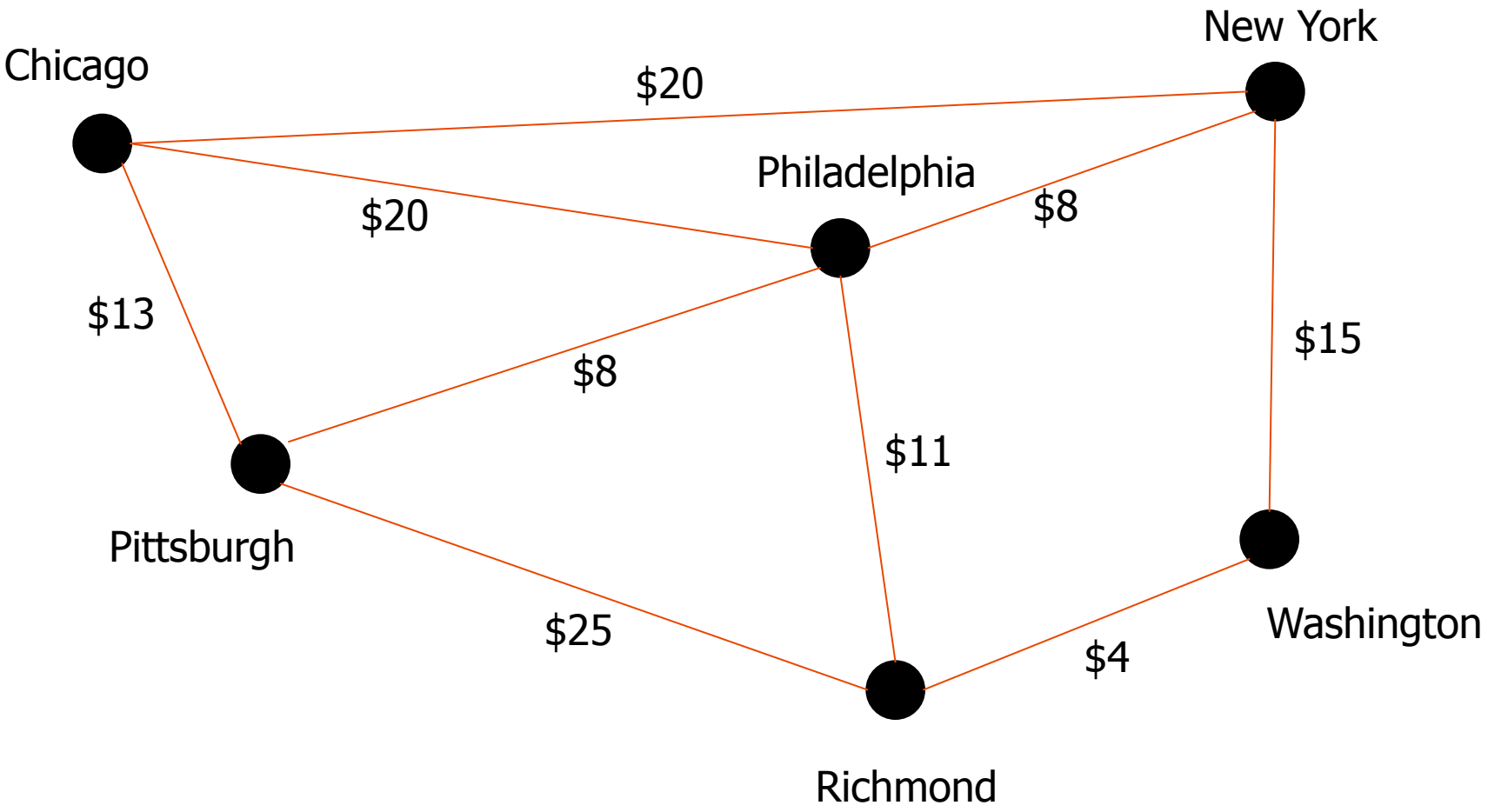


Examples of Graphs in the Wild



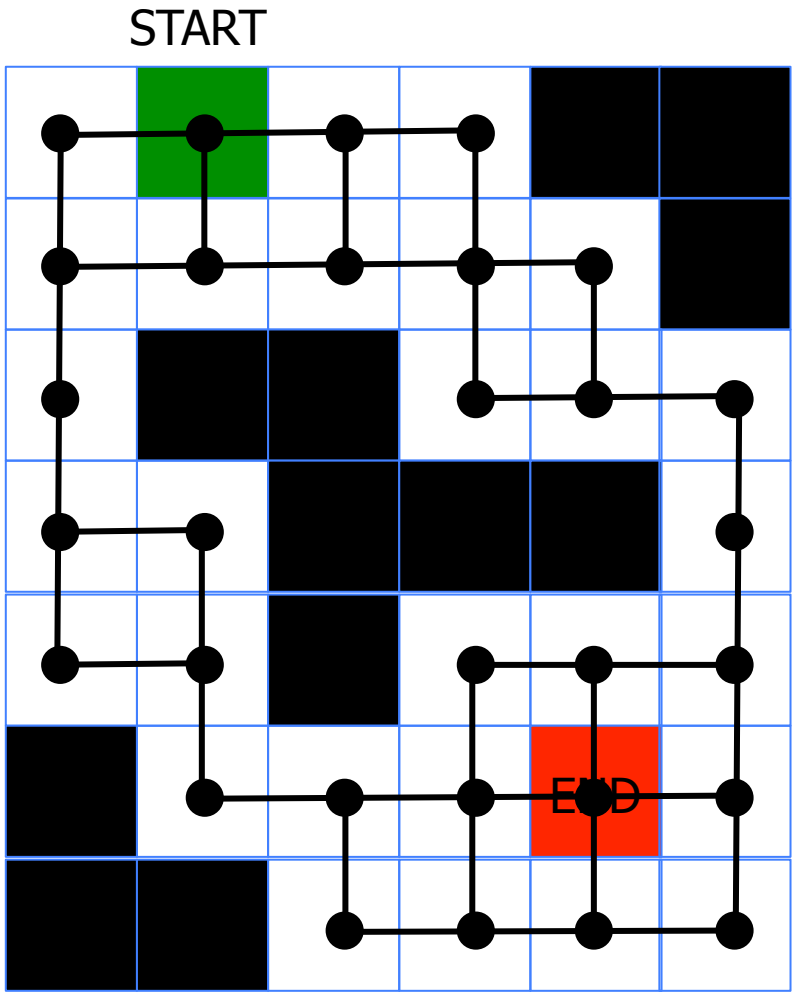
Penn

Examples of Graphs in the Wild – Toll Chart



Graph Structure

- In this grid graph we will implicitly associate a cost or distance of 1 with every edge in the graph since they link adjacent cells.



Grassfire / Breadth First Search

SECTION 1.2

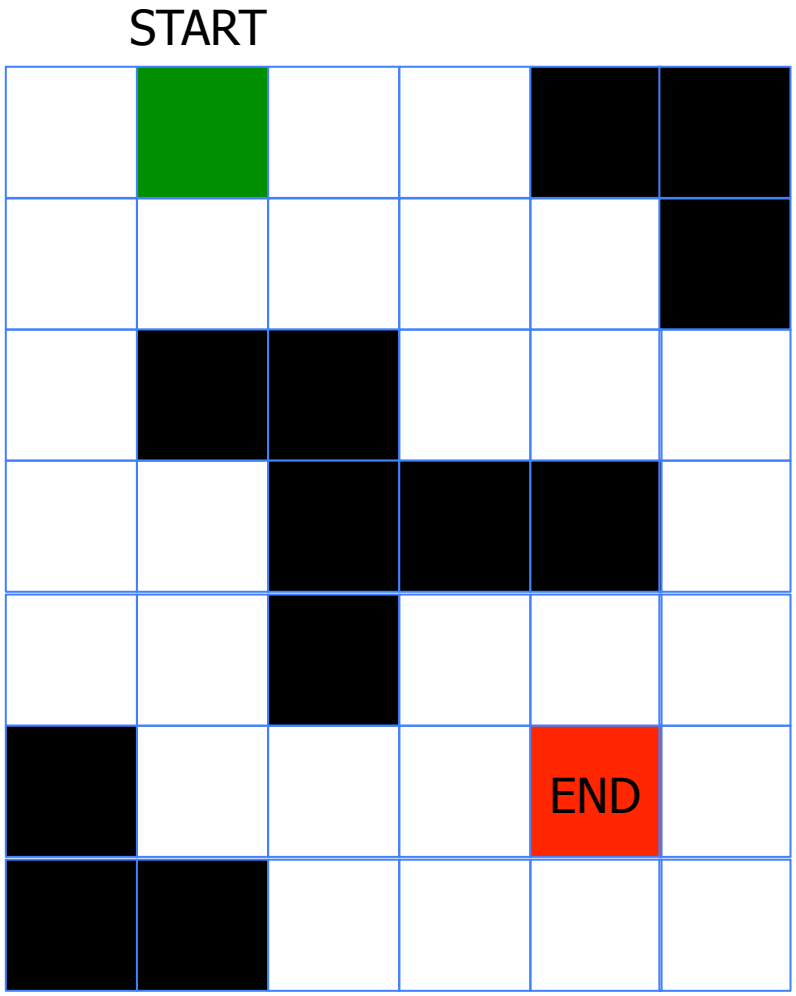
Penn

Section 1.2 - Grassfire

Prof. C.J. Taylor

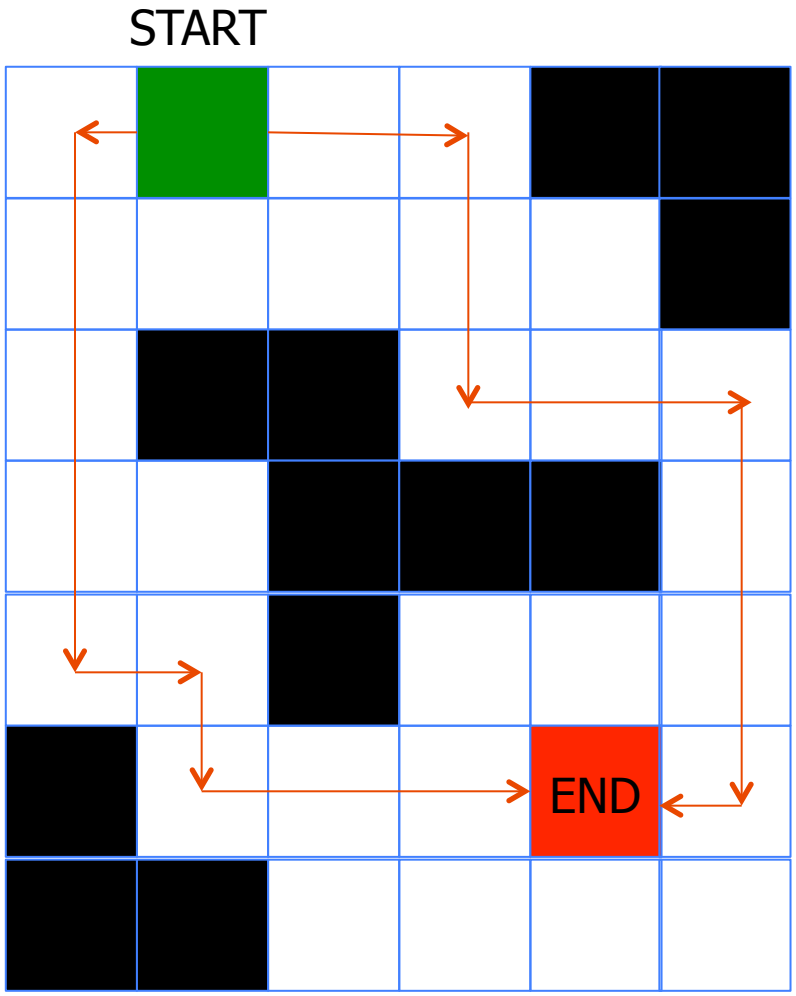
Planning on a grid

- The goal is to construct a path through the grid/ graph from the start to the goal



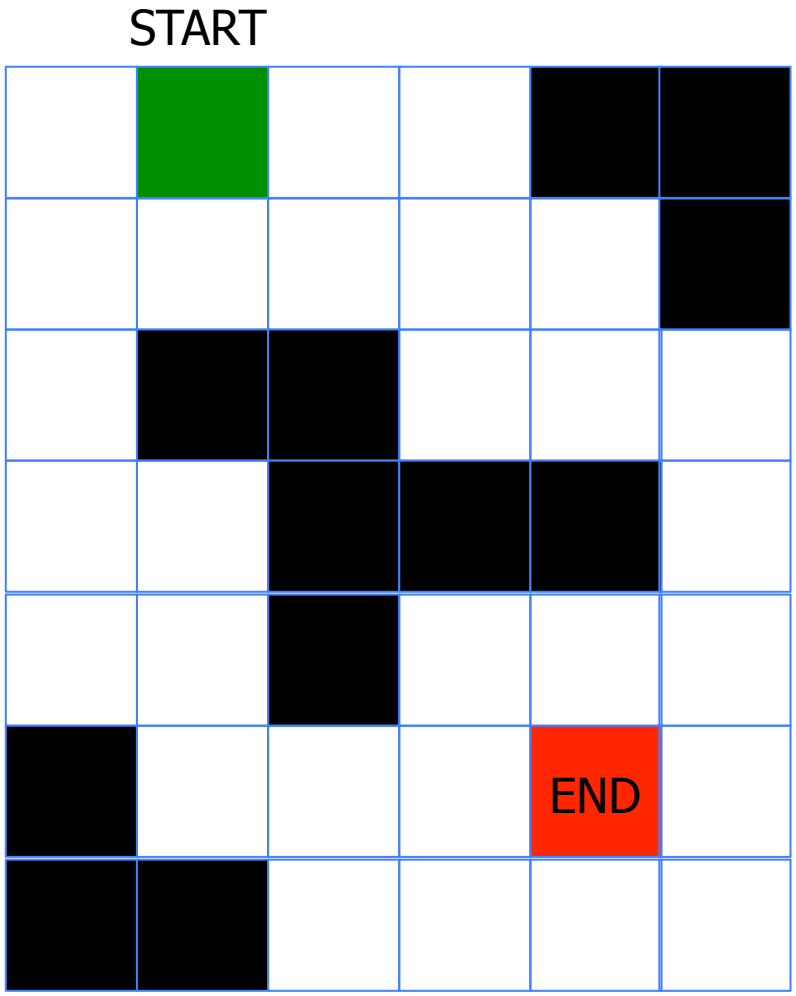
Planning on a grid

- Typically there are many possible paths between two nodes.
- We are usually interested in the shortest paths



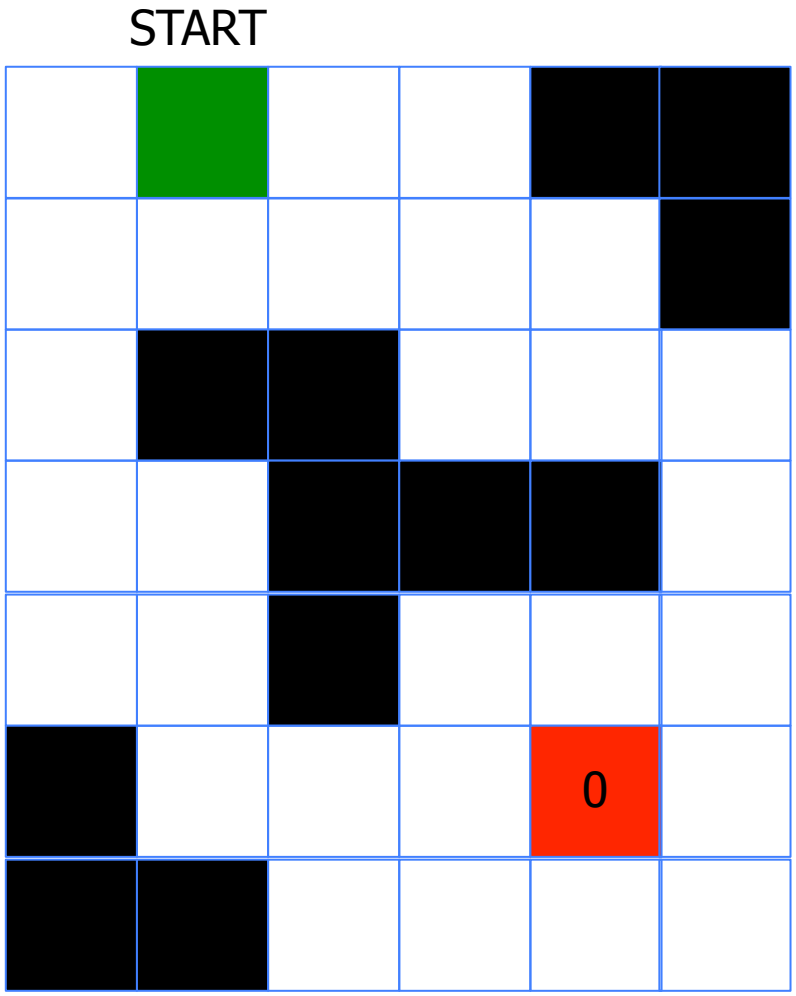
Planning on a grid

- Goal:
  - Construct the shortest path between the start and the goal location.



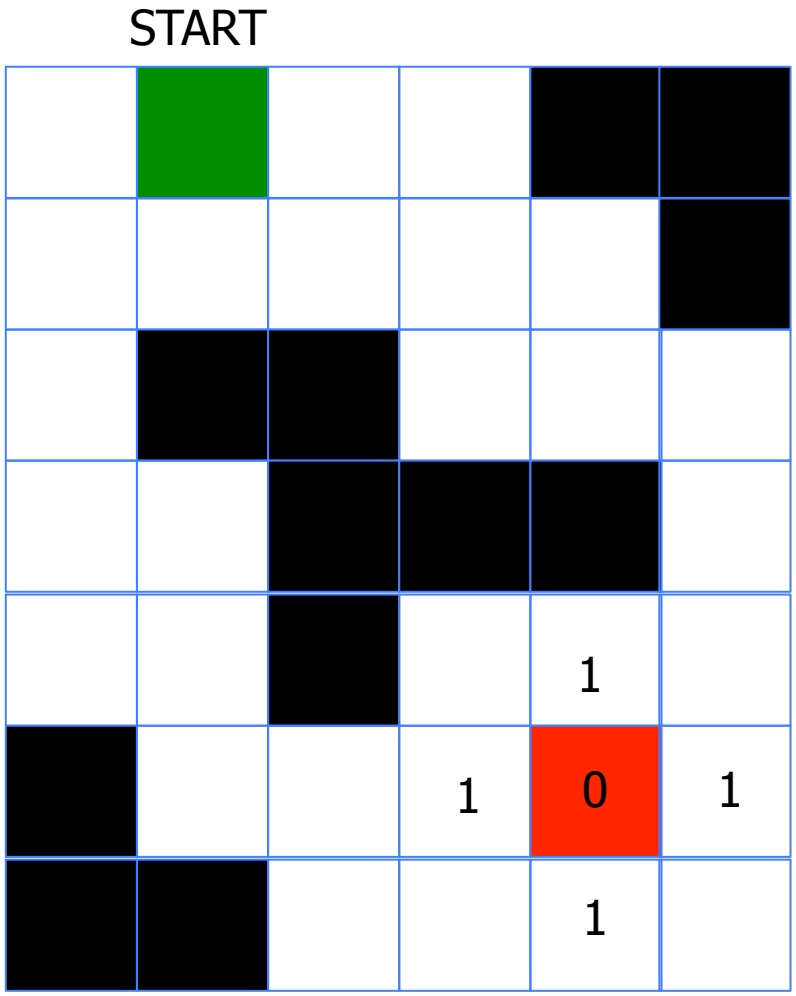
Planning Procedure – Grassfire Algorithm

- Begin by marking the destination node with a distance value of 0

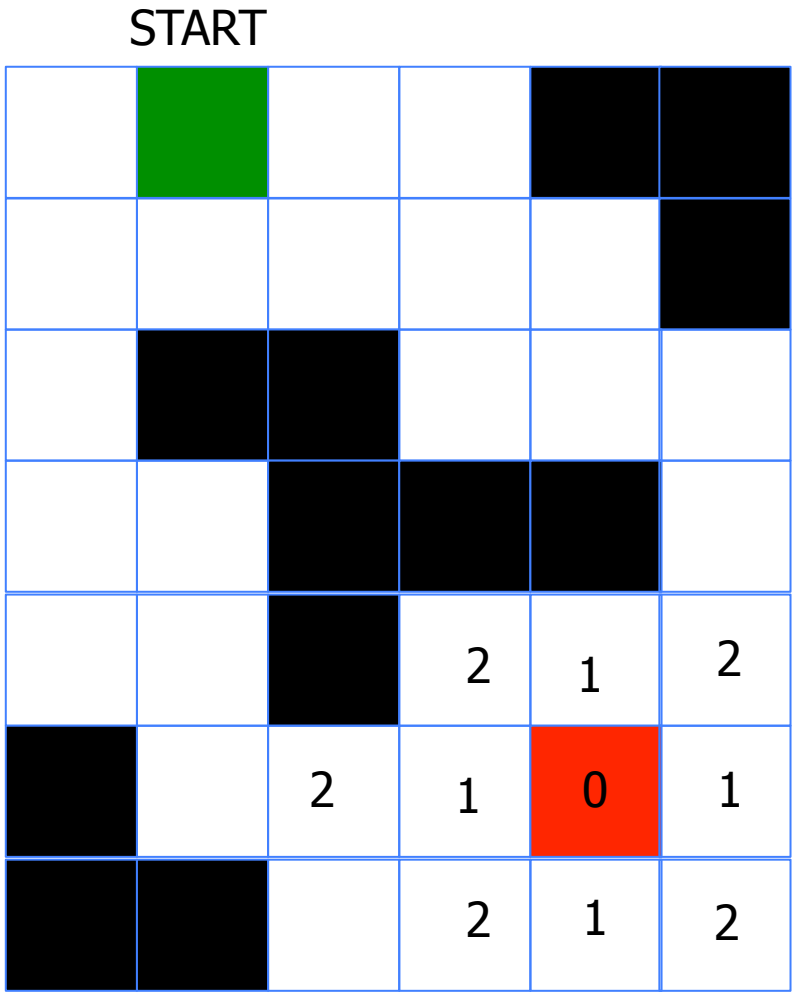


Planning Procedure – Grassfire Algorithm

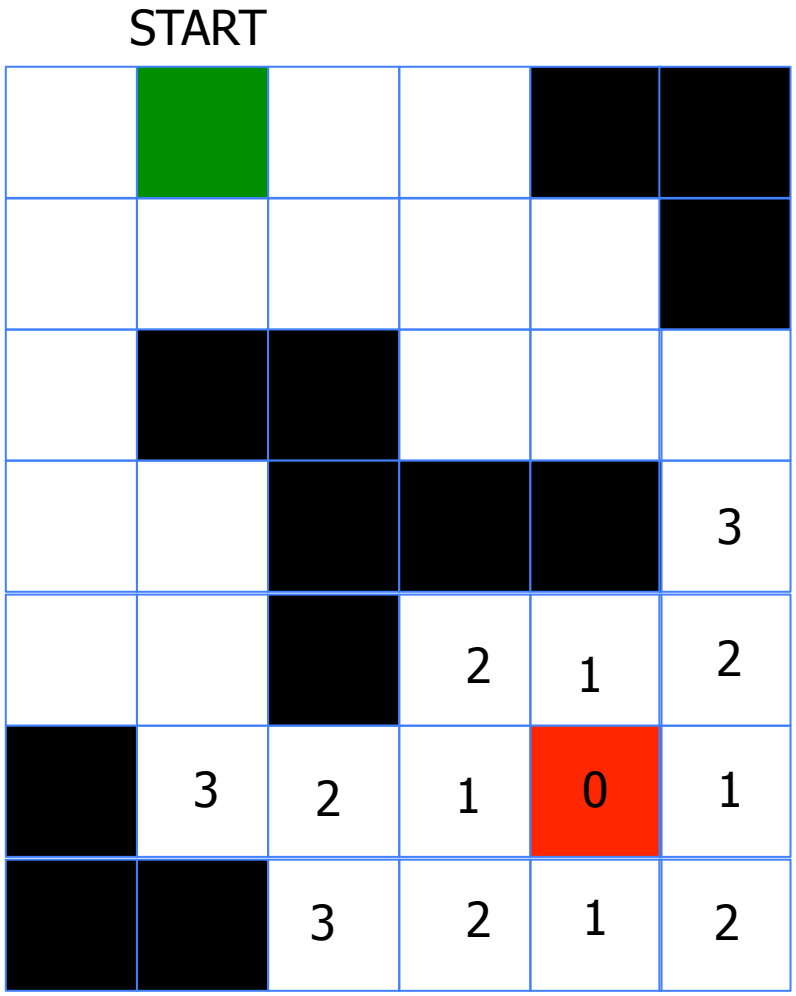
- On every iteration find all the unmarked nodes adjacent to marked nodes and mark them with that distance value + 1.



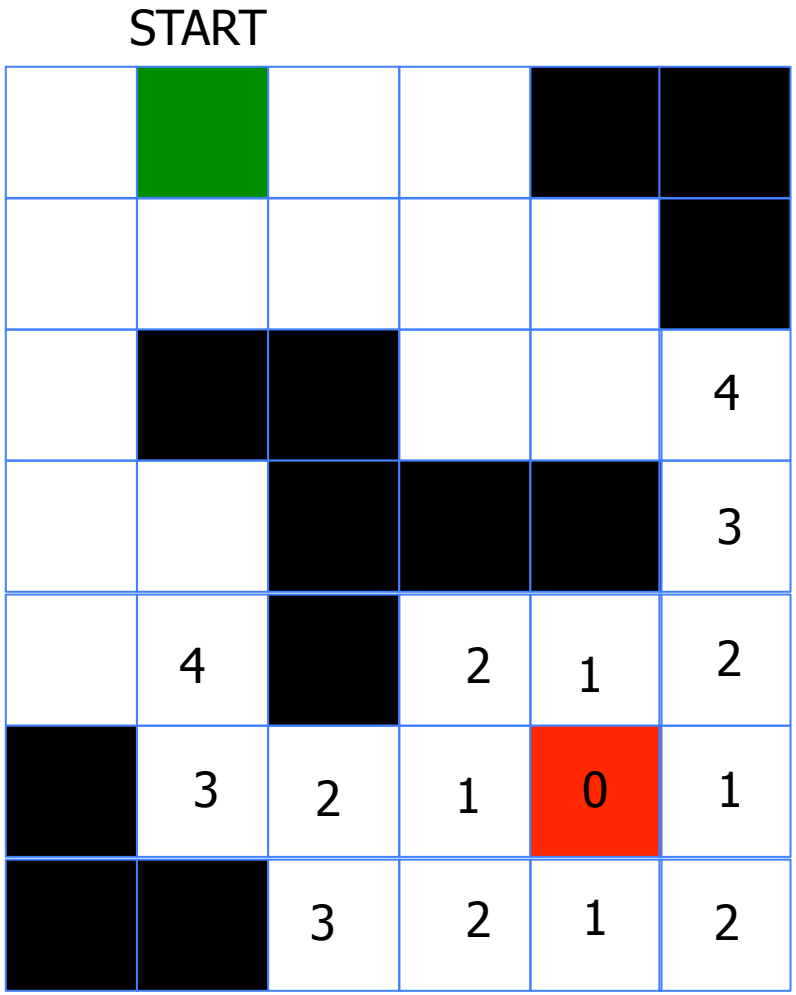
Planning Procedure – Grassfire Algorithm



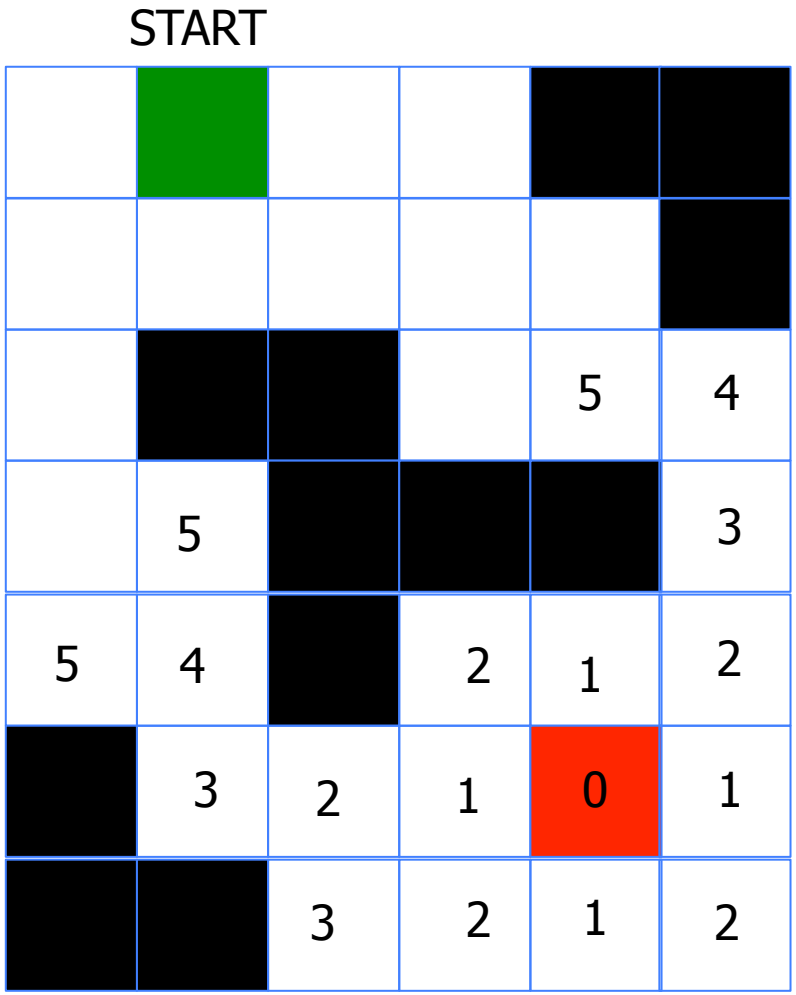
Planning Procedure – Grassfire Algorithm



Planning Procedure – Grassfire Algorithm

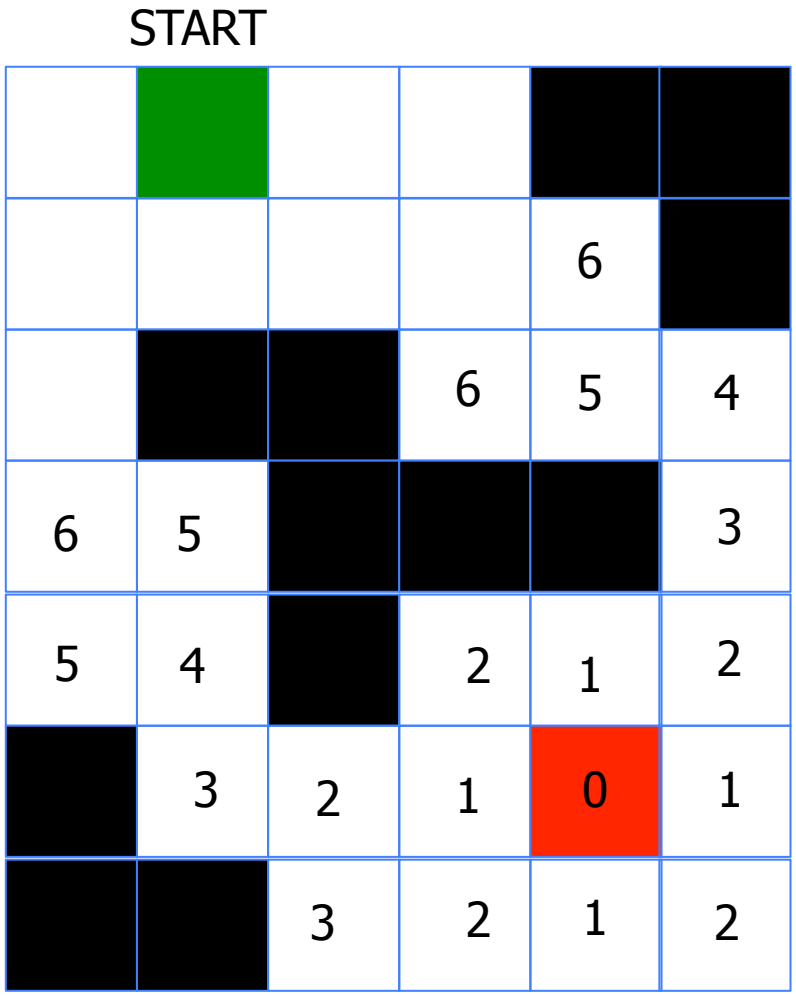


Planning Procedure – Grassfire Algorithm

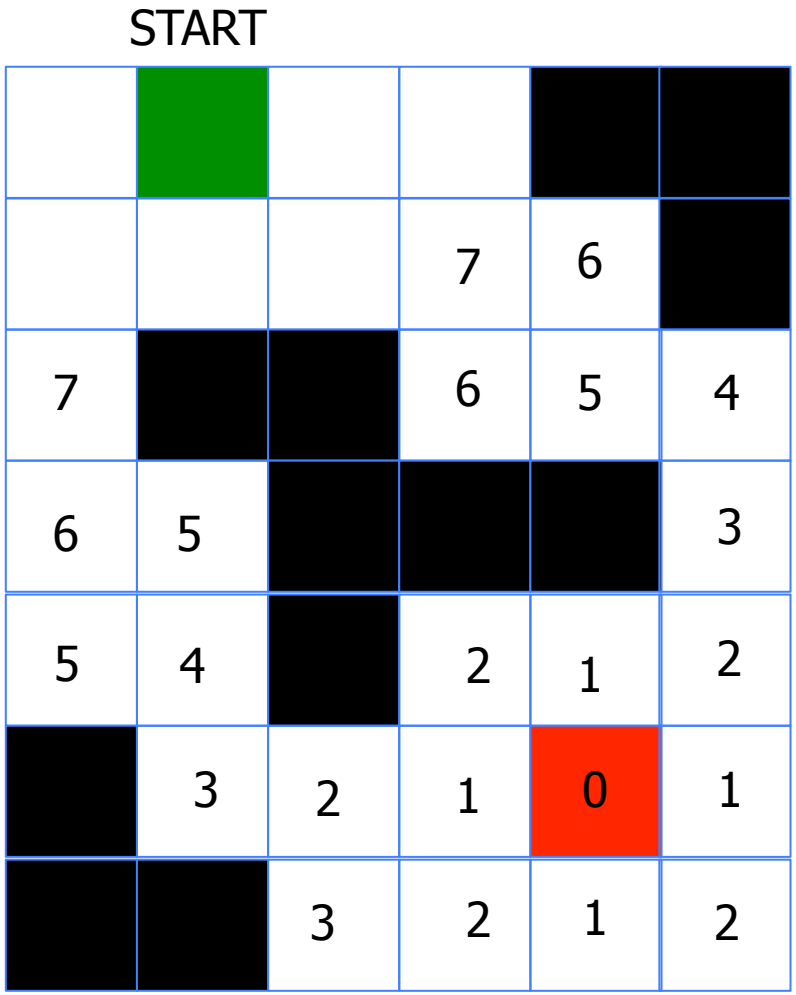


Planning Procedure – Grassfire Algorithm

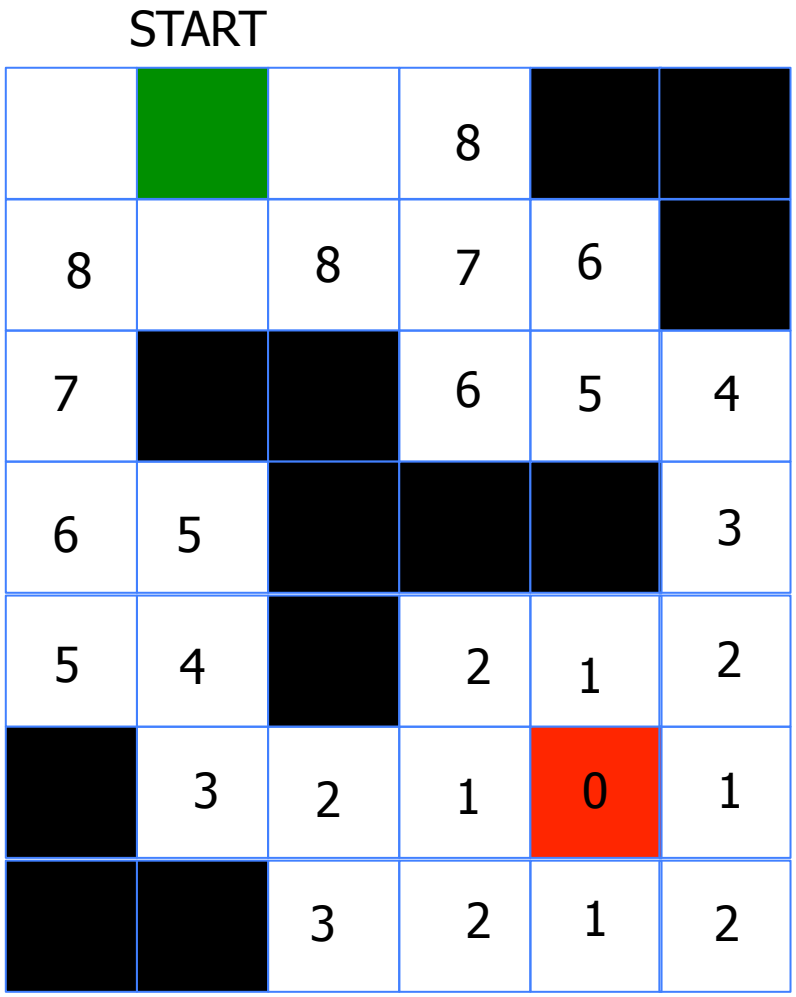
- On every iteration the marking radiates outward from the destination like a fire spreading – hence the name



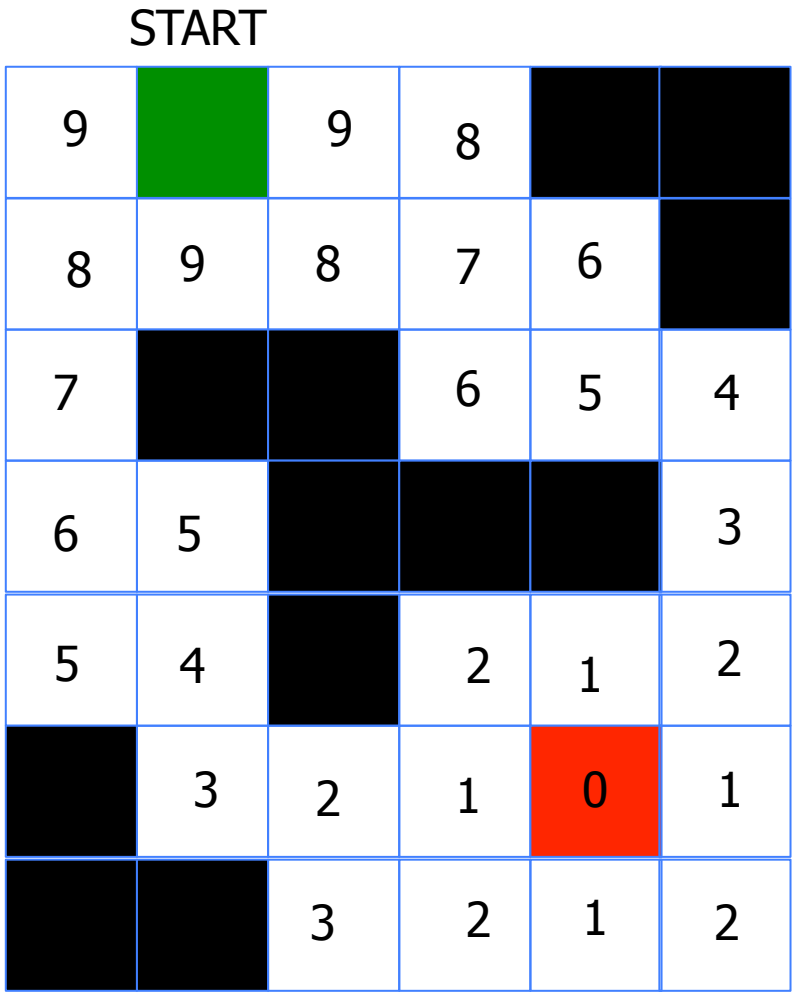
Planning Procedure – Grassfire Algorithm



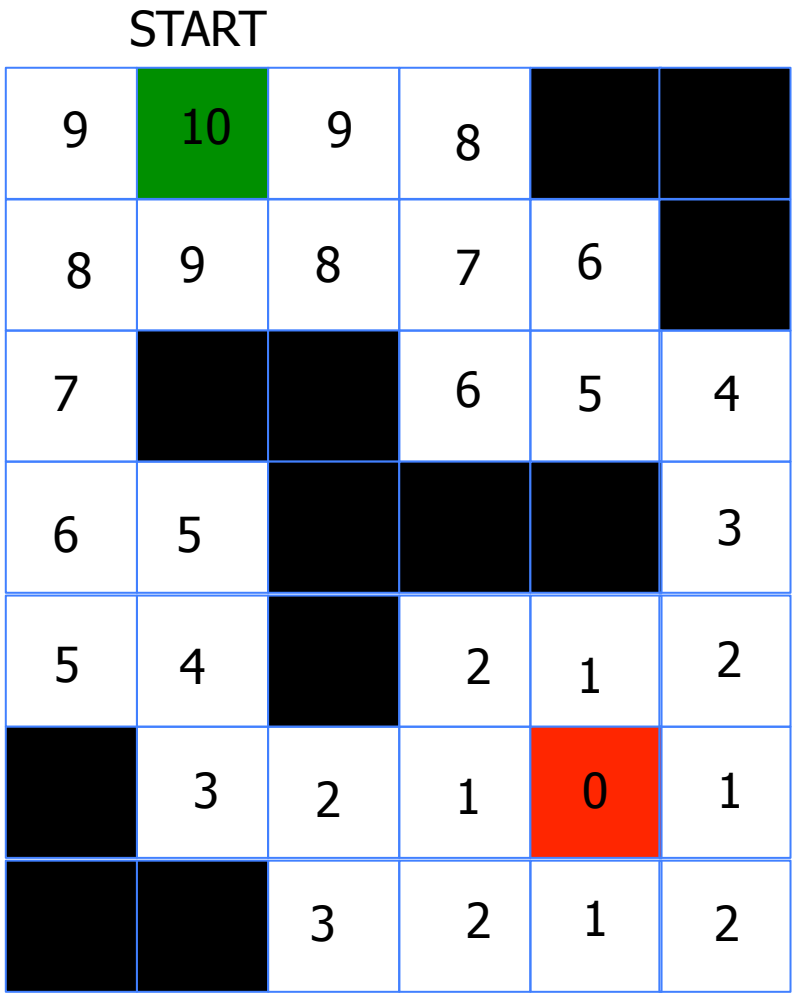
Planning Procedure – Grassfire Algorithm



Planning Procedure – Grassfire Algorithm

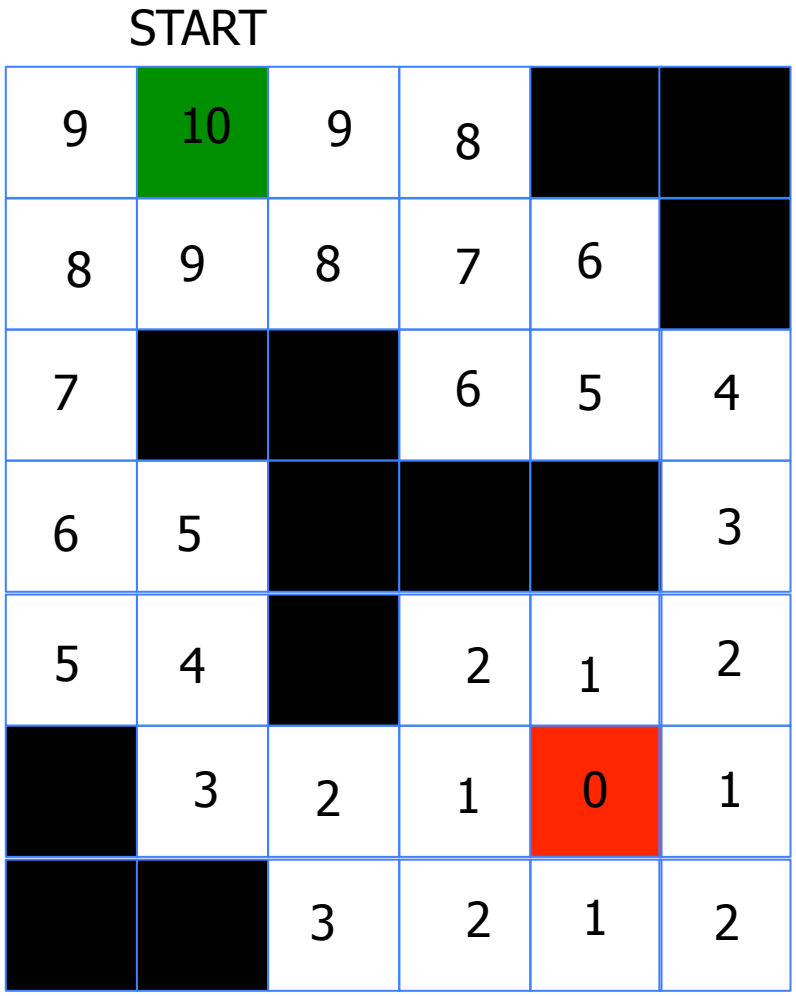


Planning Procedure – Grassfire Algorithm



Planning Procedure – Grassfire Algorithm

- The distance values produced by the grassfire algorithm indicate the smallest number of steps needed to move from each node to the goal



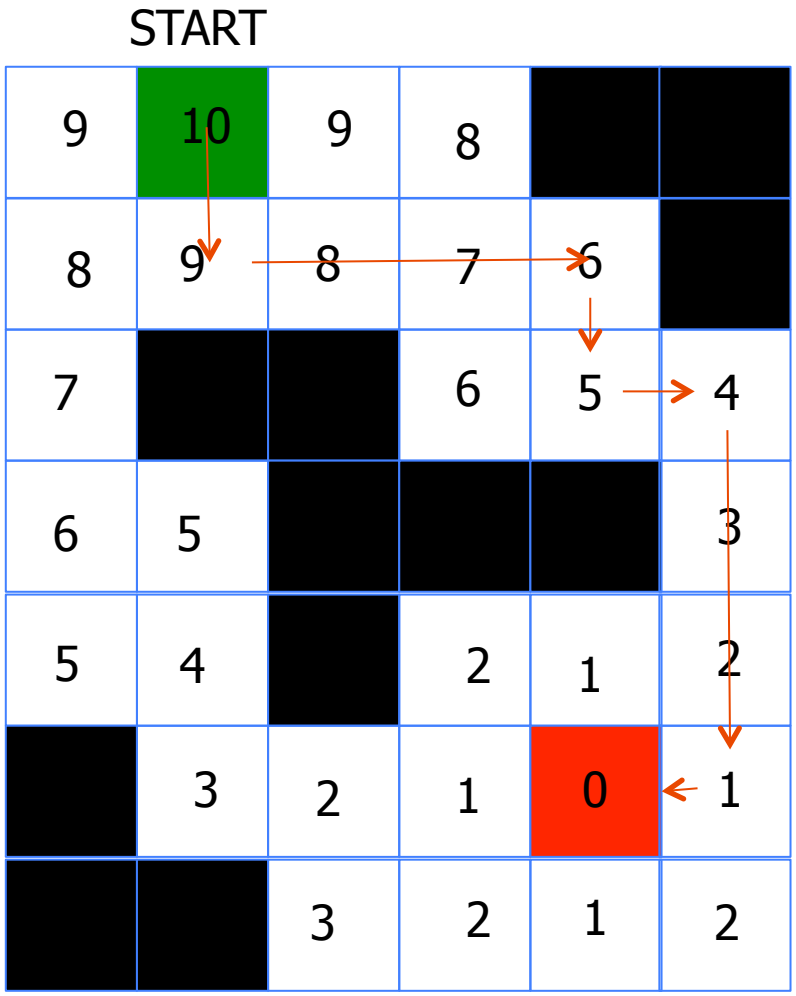
Grassfire algorithm – pseudo code

- For each node n in the graph
  - n.distance = Infinity
- Create an empty list.
- goal.distance = 0, add goal to list.
- While list not empty
  - Let current = first node in list, remove current from list
  - For each node, n that is adjacent to current

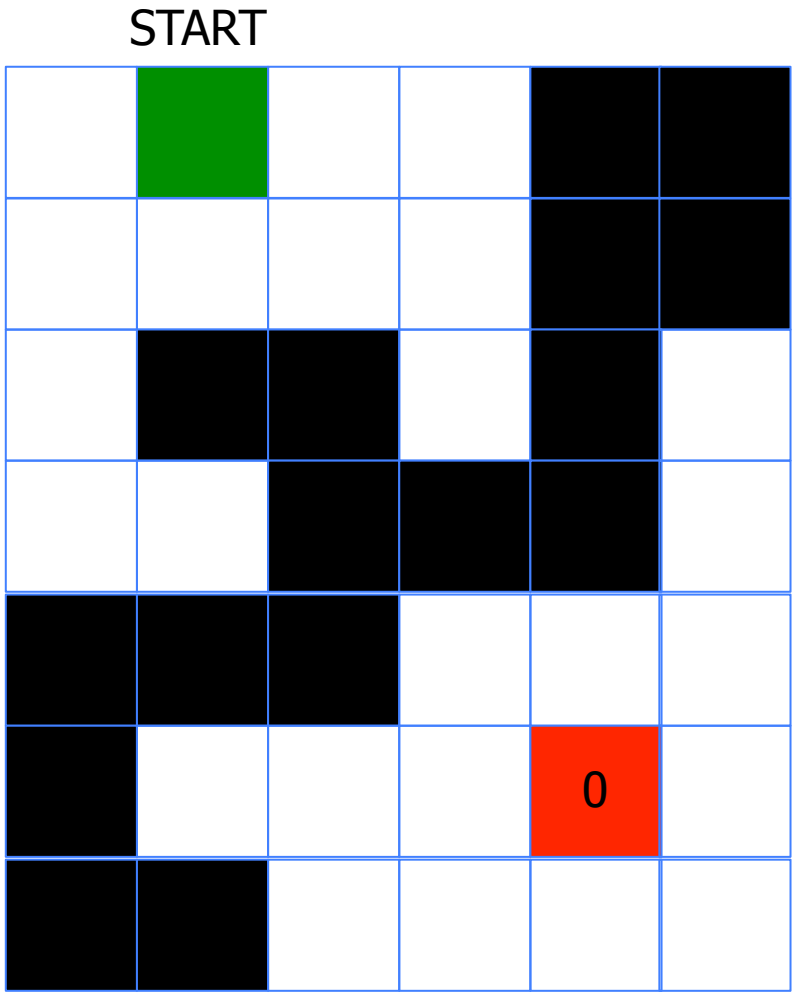
```
If n.distance = Infinity
    n.distance = current.distance + 1
    add n to the back of the list
```

Tracing a path to the destination

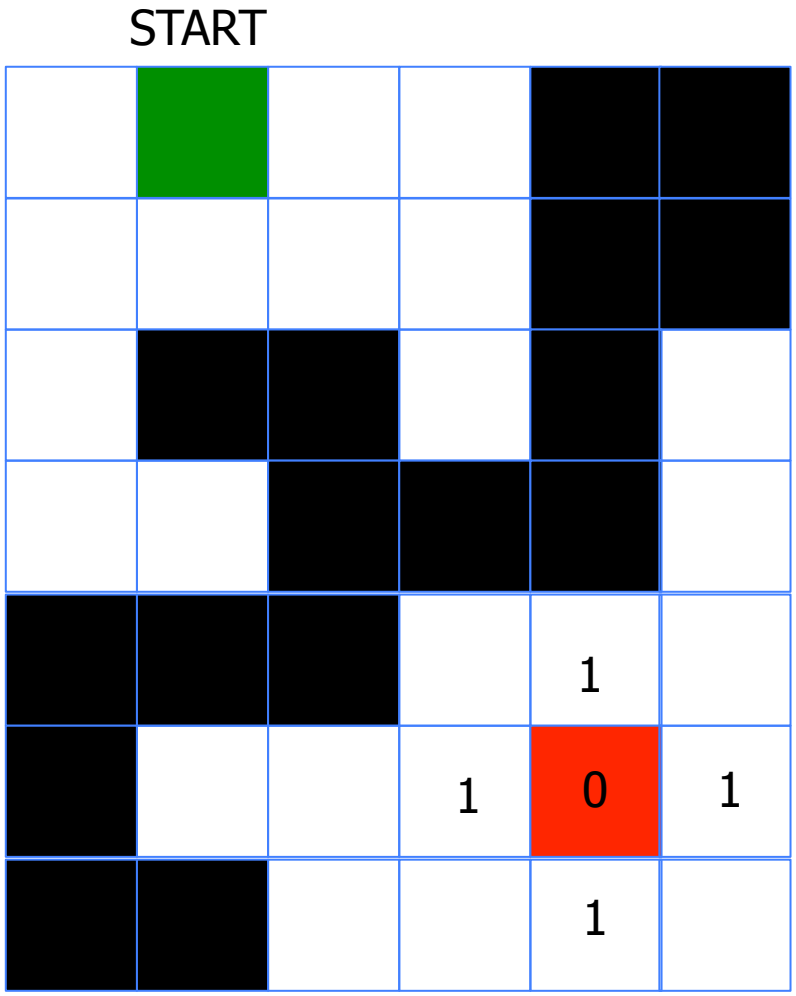
- To move towards the destination from any node simply move towards the neighbor with the smallest distance value, breaking ties arbitrarily.



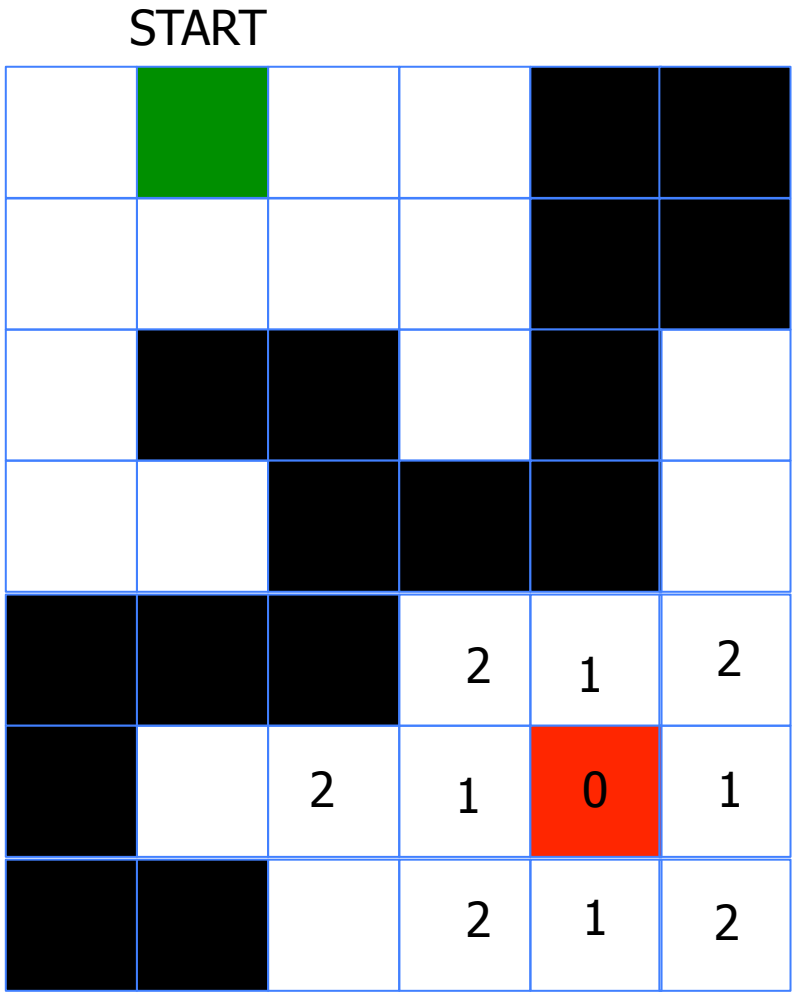
Another Example – Grassfire Algorithm



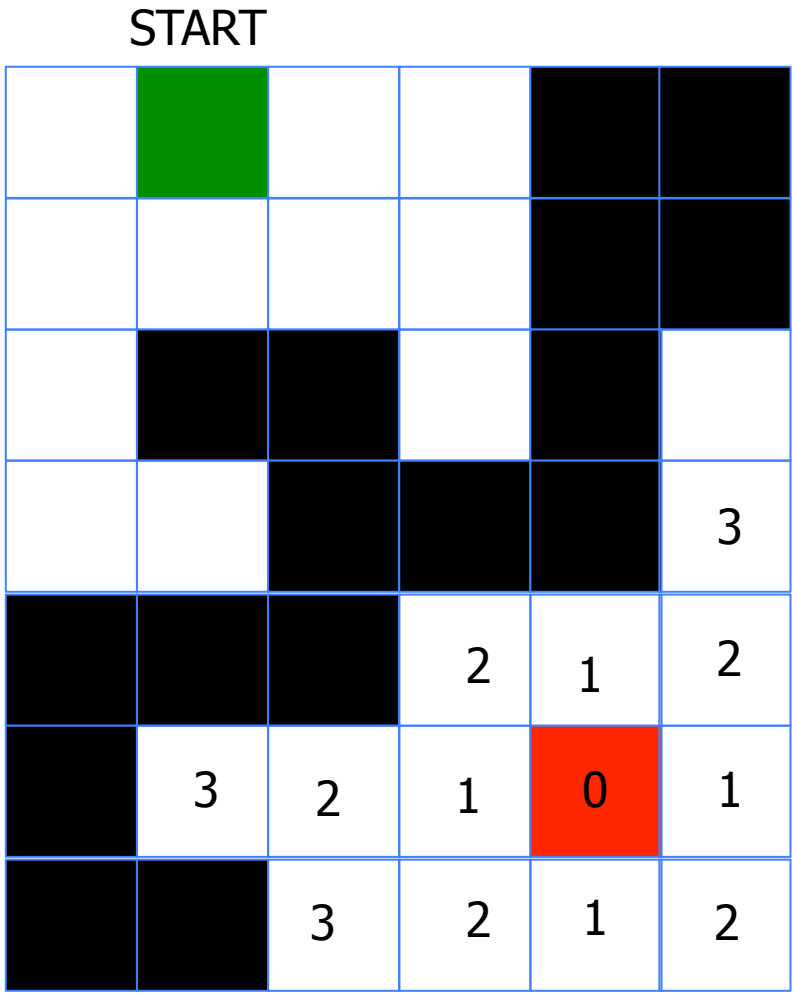
Another Example – Grassfire Algorithm



Planning Procedure – Grassfire Algorithm

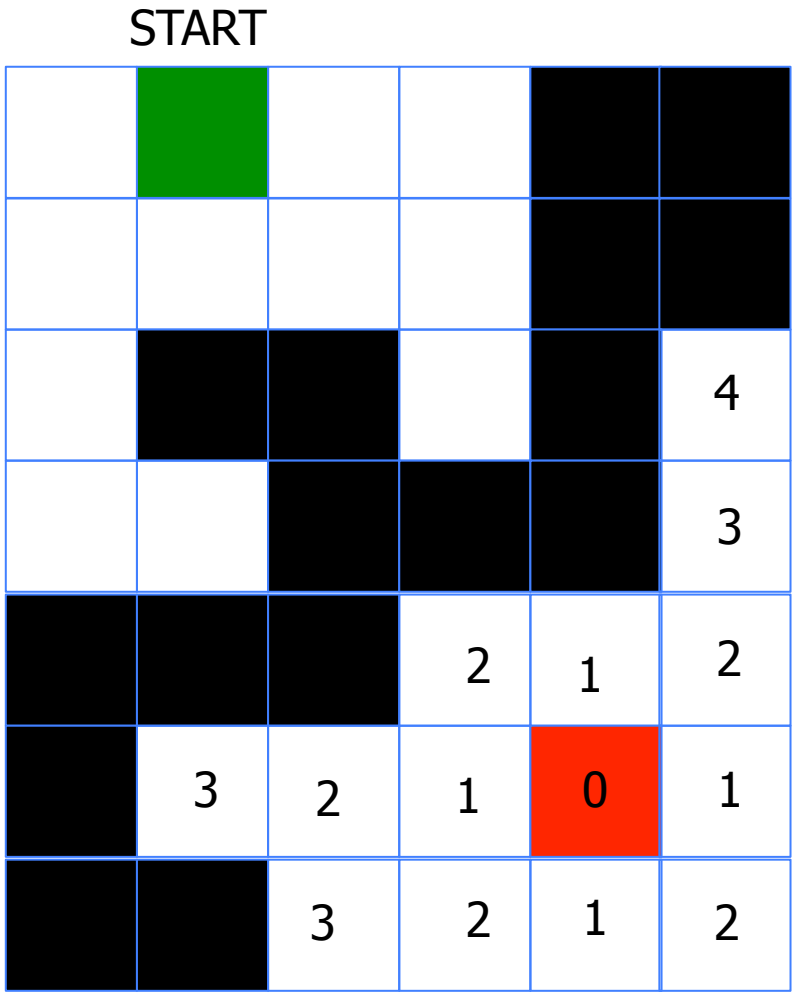


Planning Procedure – Grassfire Algorithm



Planning Procedure – Grassfire Algorithm

- In this case the procedure terminates before the start node is marked indicating that no path exists



Grassfire Algorithm

- It will find the shortest path between the start and the goal if one exists.
- If no path exists that fact will be discovered.

Computational Complexity - Grassfire

- The computational effort required to run the grassfire algorithm on a grid increases linearly with the number of edges.
- This can be expressed more formally as follows.

$$\mathcal{O}(|\mathbf{V}|) \tag{1}$$

Where $|\mathbf{V}|$ denotes the number of nodes in the graph

Computational Complexity - Grassfire

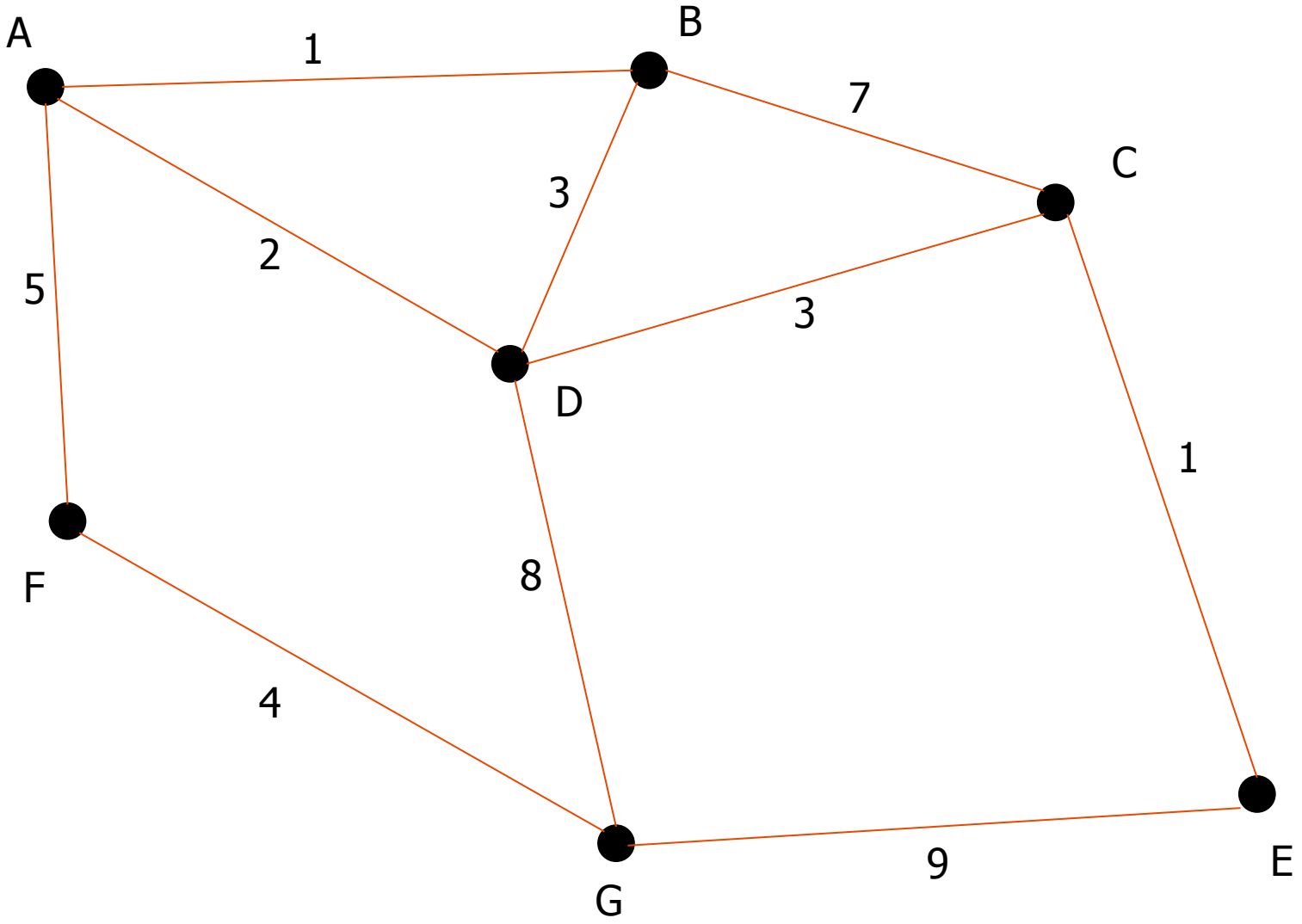
- Number of nodes in a 2D grid 100x100 = 10^4
- Number of nodes in a 3D grid 100x100x100 = 10^6
- Number of nodes in a 6D grid 100 cells on side = 10^{12}

Dijkstra's Algorithm

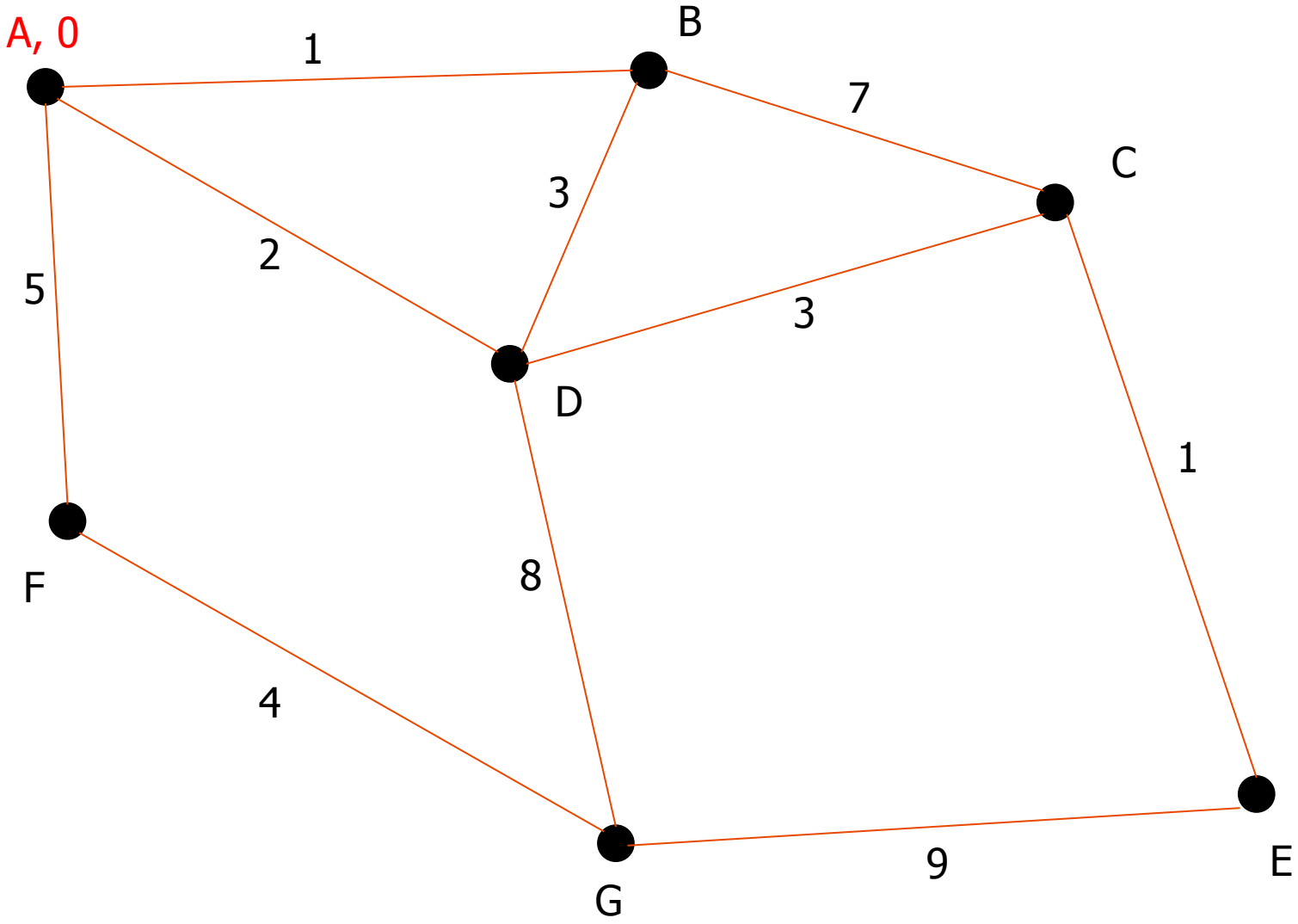
SECTION 1.3

Penn

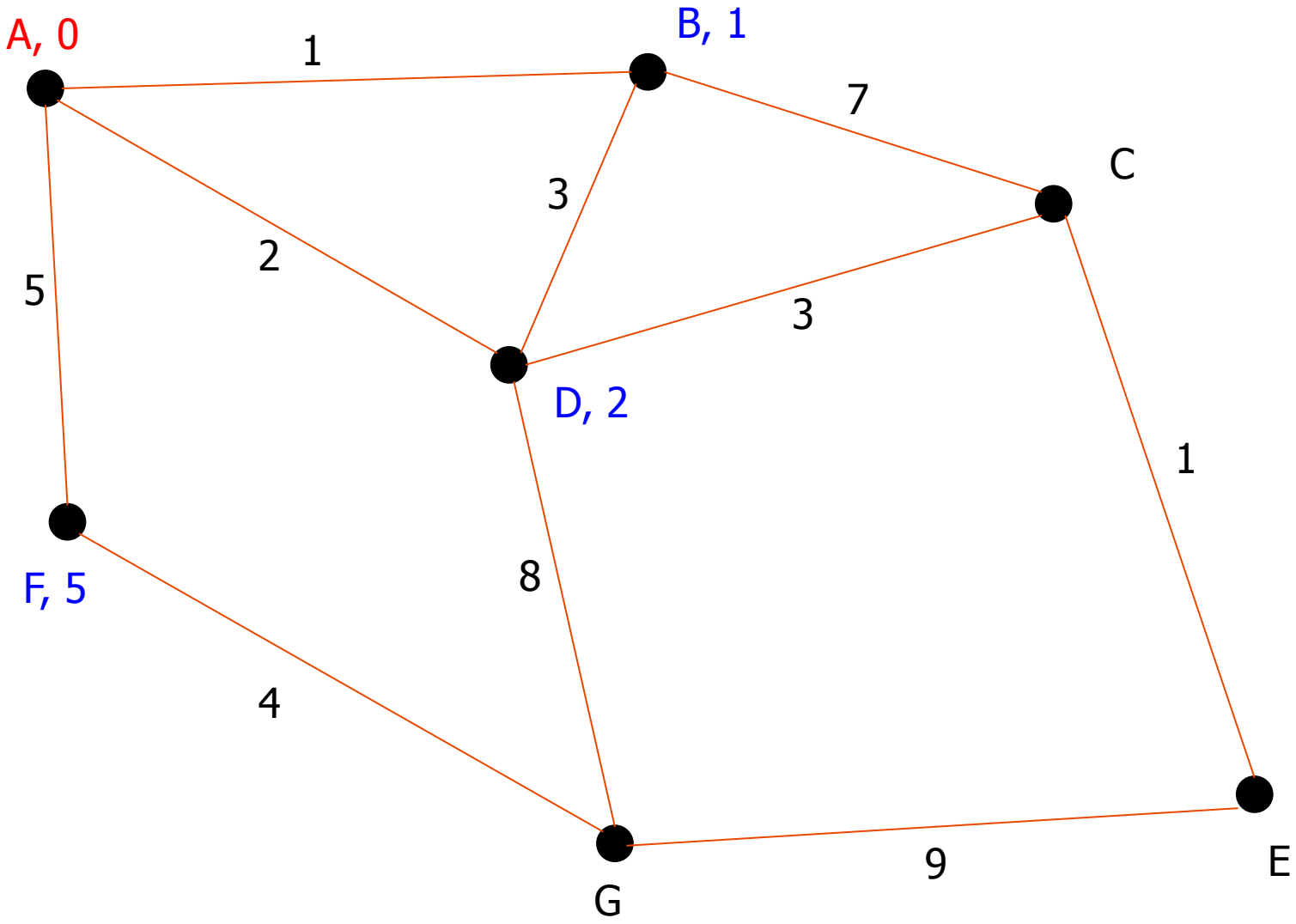
Planning shortest paths – Dijkstra's Algorithm



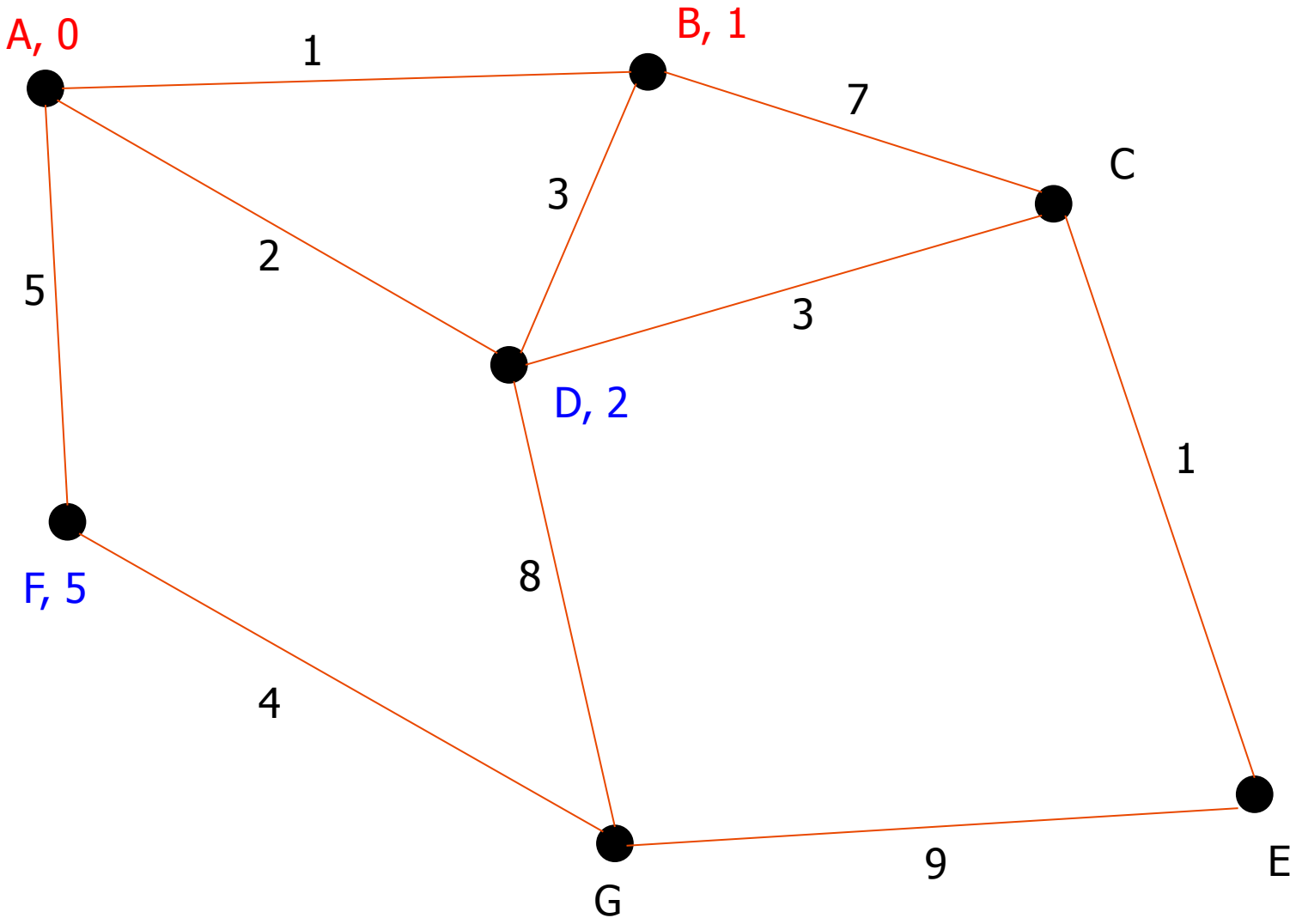
Planning shortest paths – Dijkstra's Algorithm



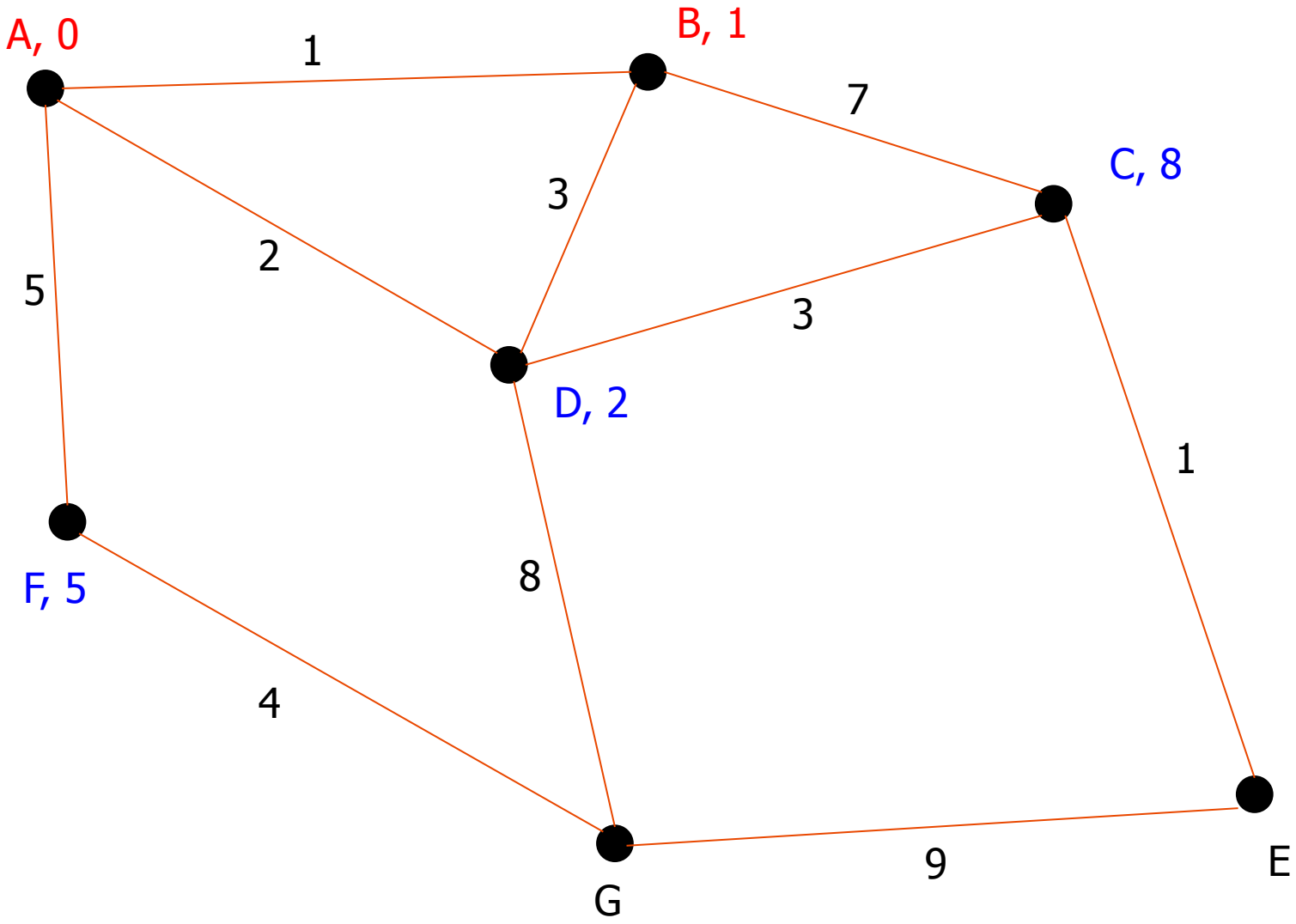
Planning shortest paths – Dijkstra's Algorithm



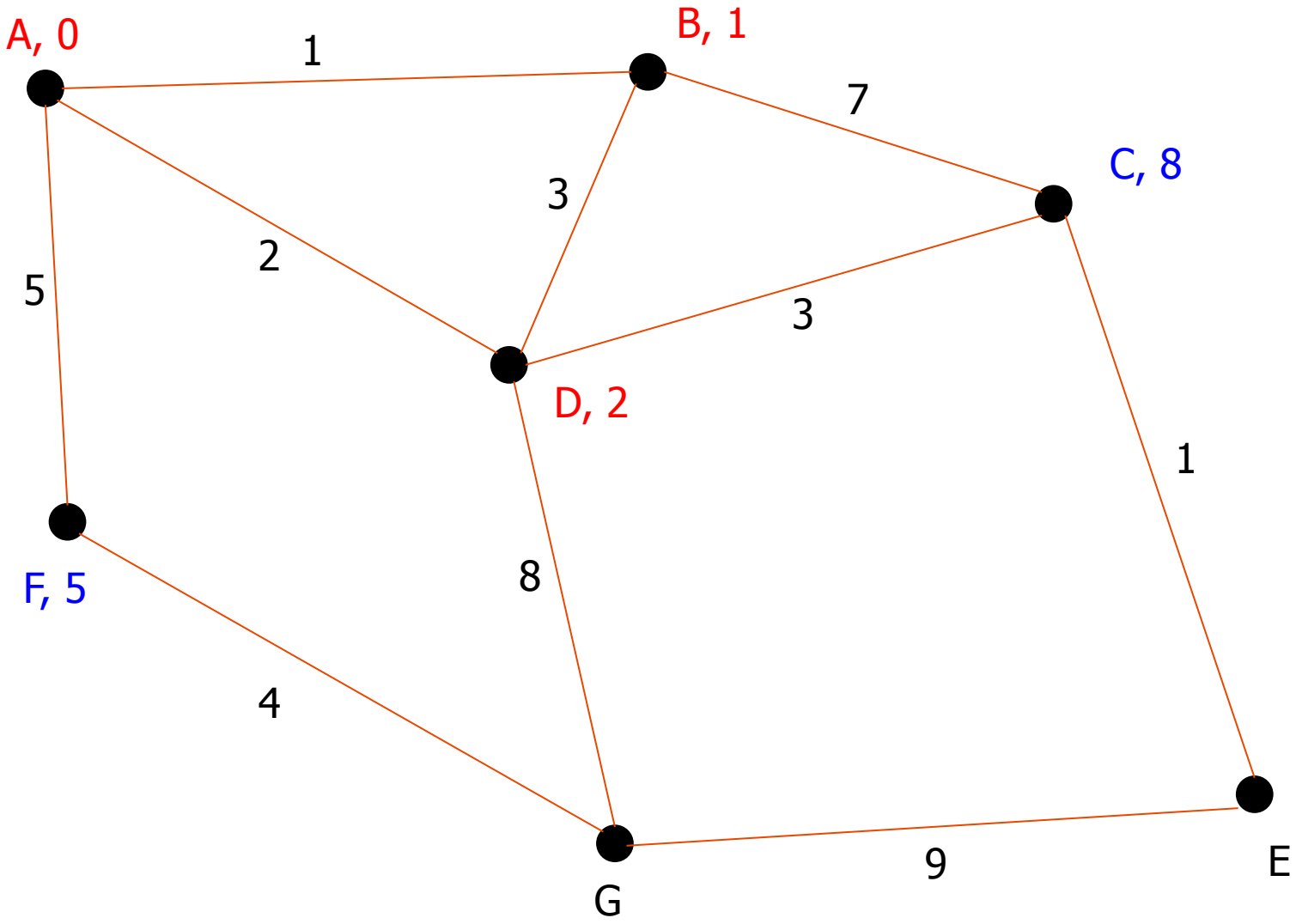
Planning shortest paths – Dijkstra's Algorithm



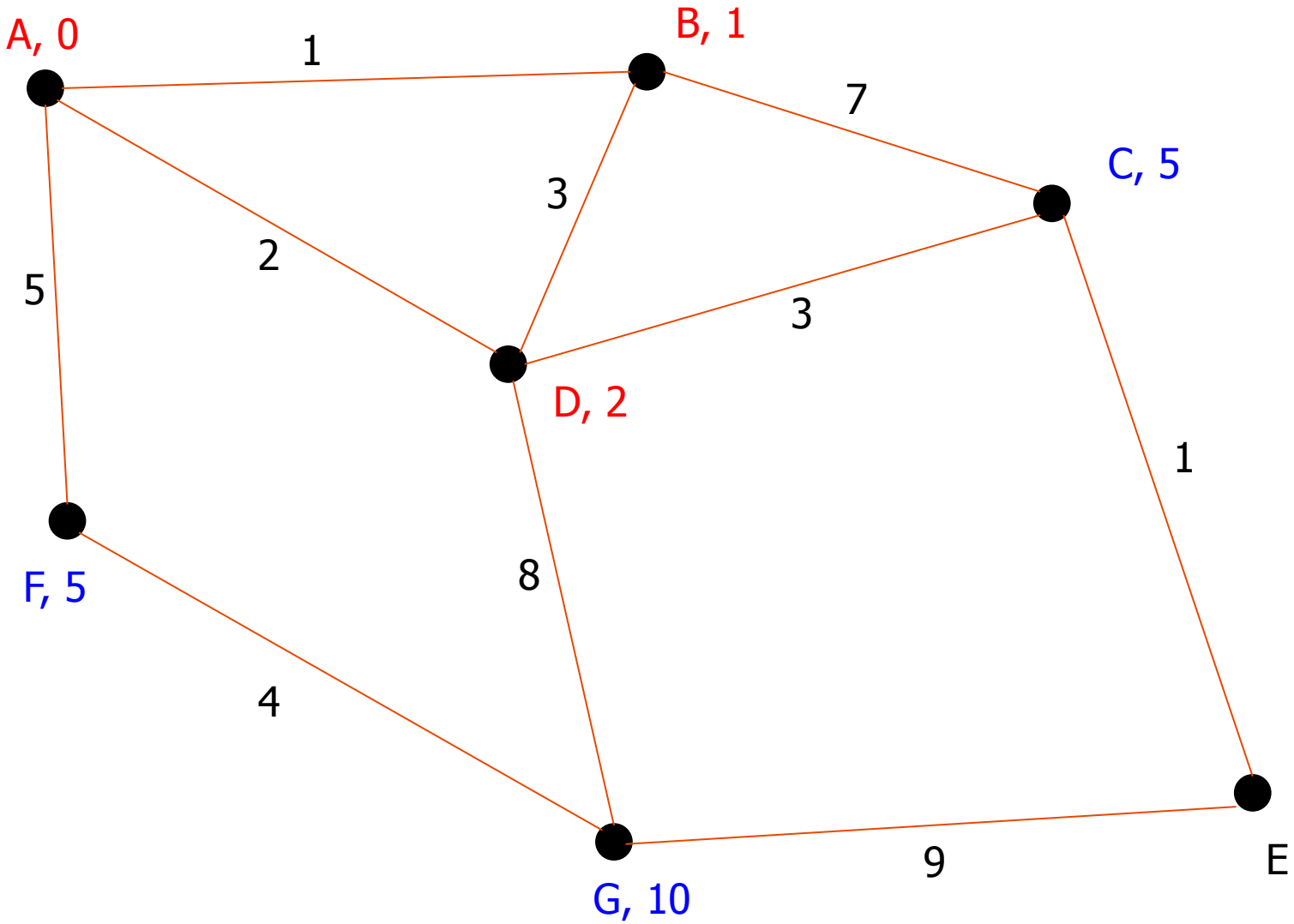
Planning shortest paths – Dijkstra's Algorithm



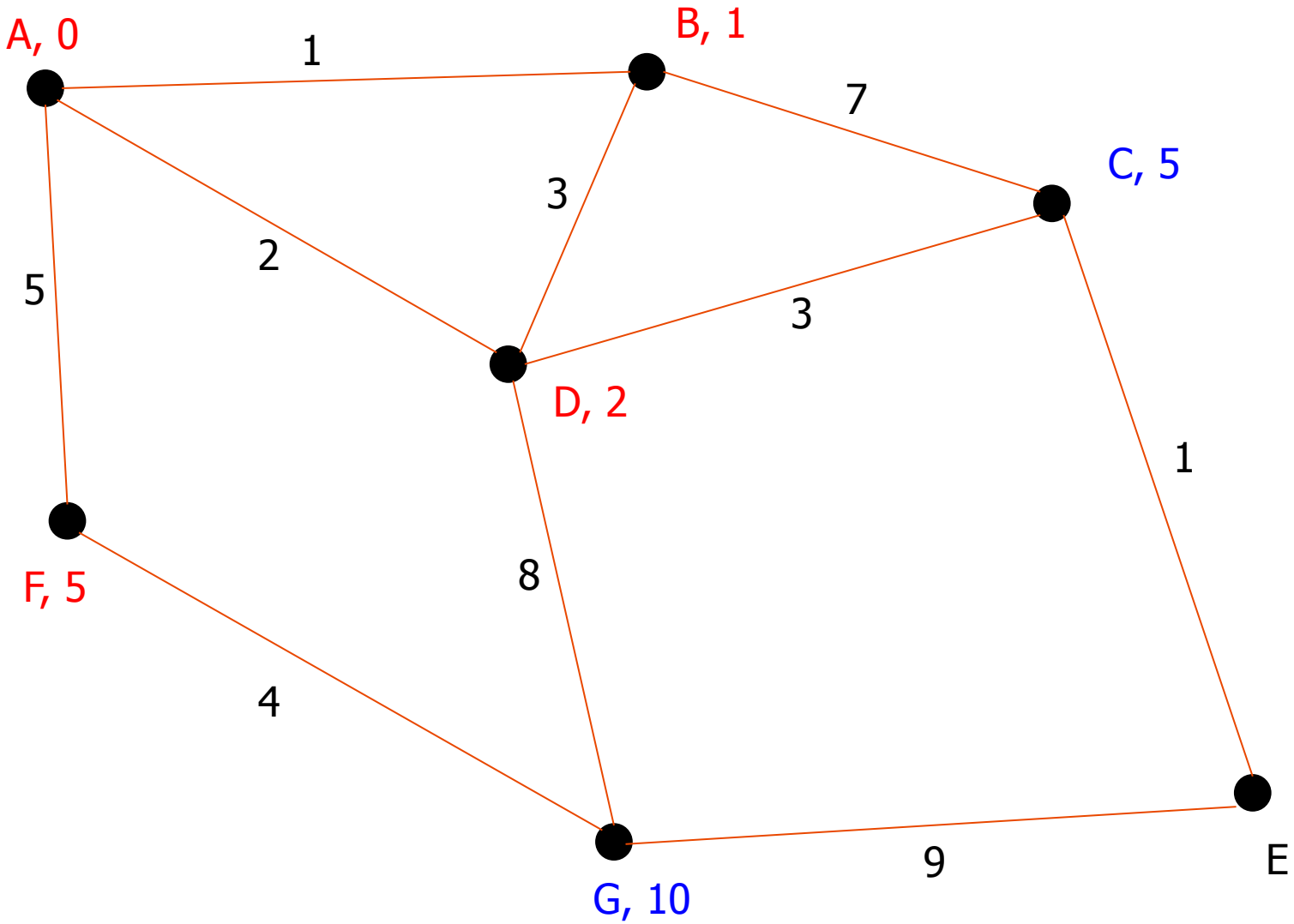
Planning shortest paths – Dijkstra's Algorithm



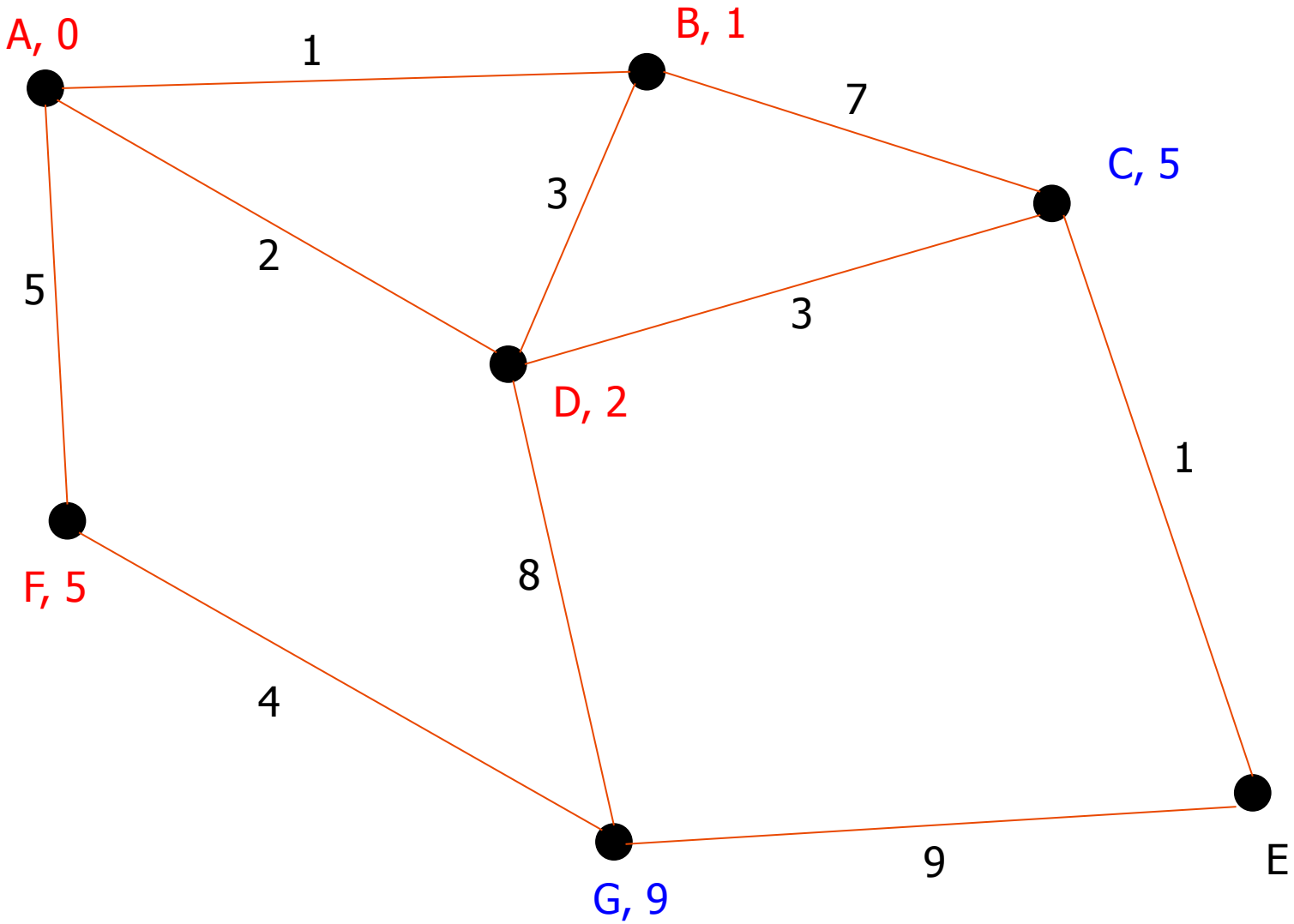
Planning shortest paths – Dijkstra's Algorithm



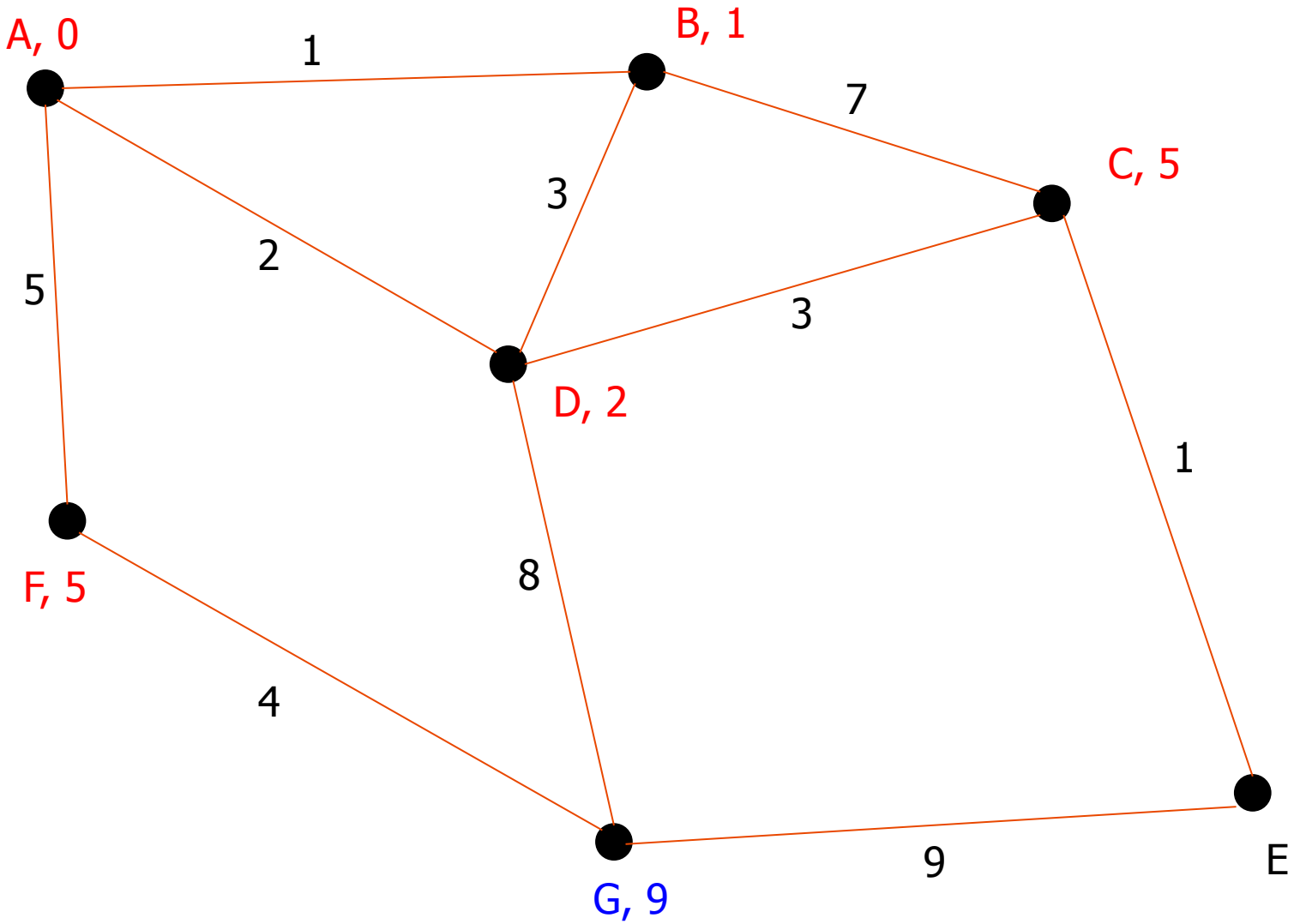
Planning shortest paths – Dijkstra's Algorithm



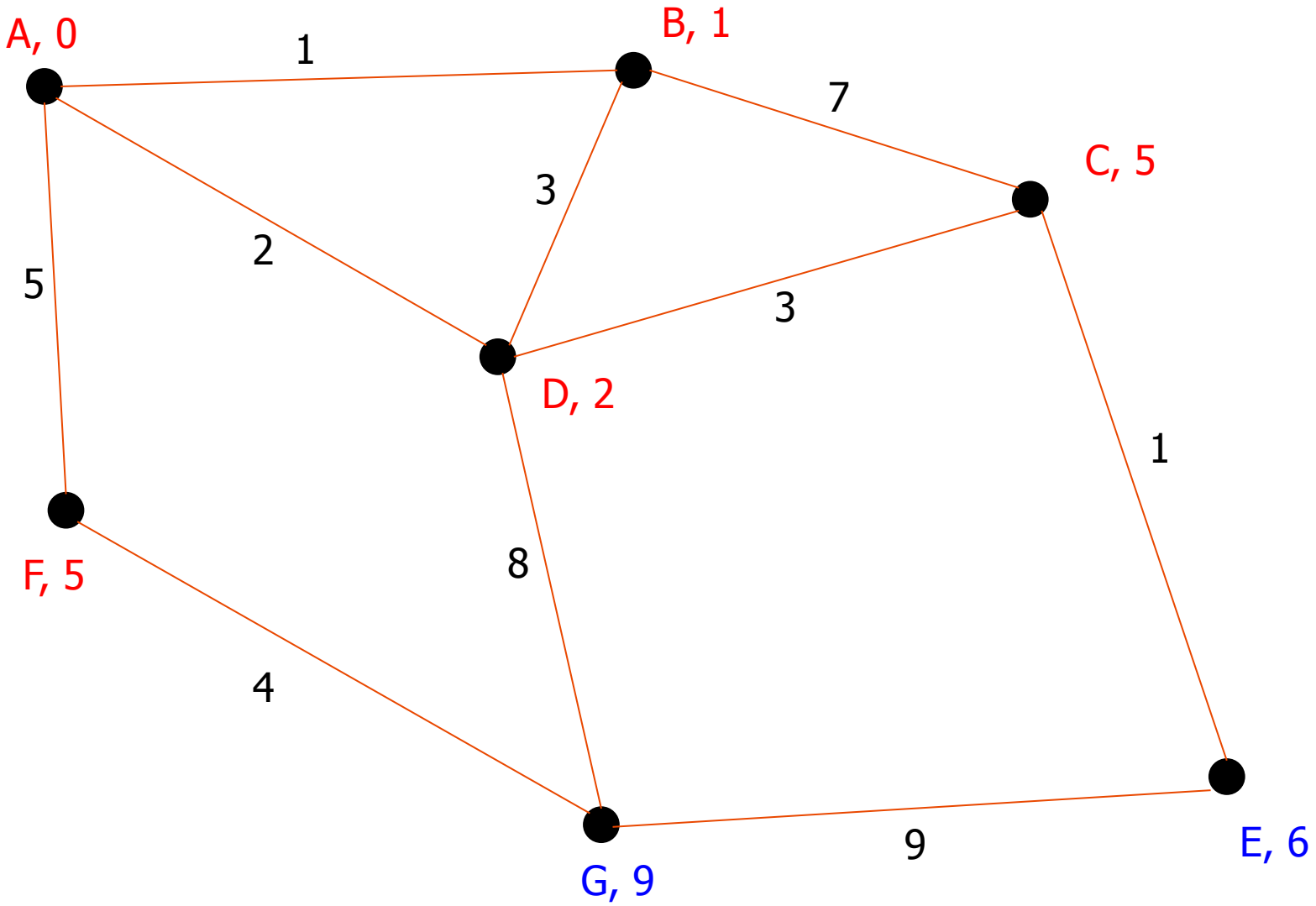
Planning shortest paths – Dijkstra's Algorithm



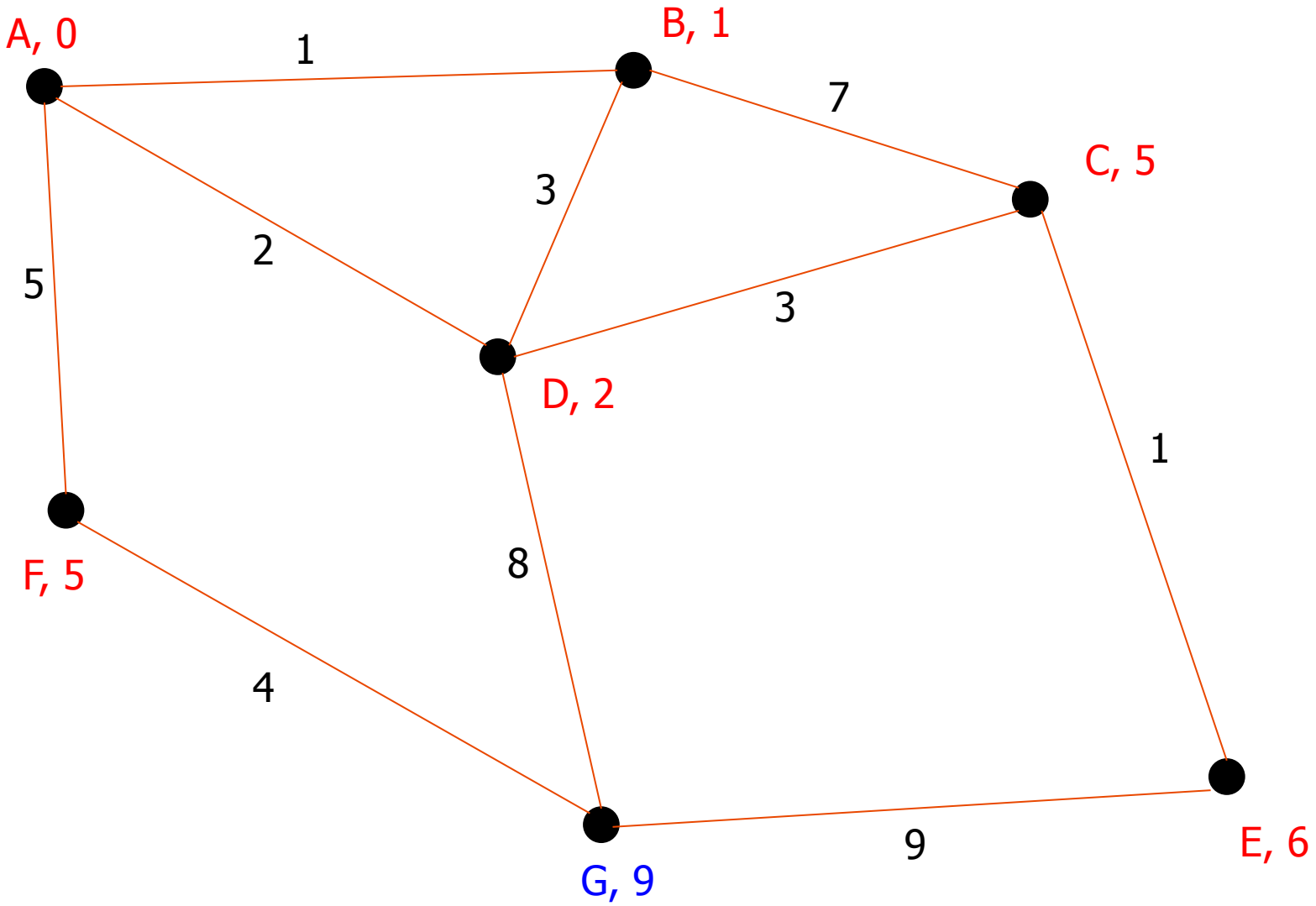
Planning shortest paths – Dijkstra's Algorithm



Planning shortest paths – Dijkstra's Algorithm



Planning shortest paths – Dijkstra's Algorithm



Dijkstra's algorithm – pseudo code

- For each node n in the graph
  - n.distance = Infinity
- Create an empty list.
- start.distance = 0, add start to list.
- While list not empty
  - Let current = node in the list with the smallest distance, remove current from list
  - For each node, n that is adjacent to current

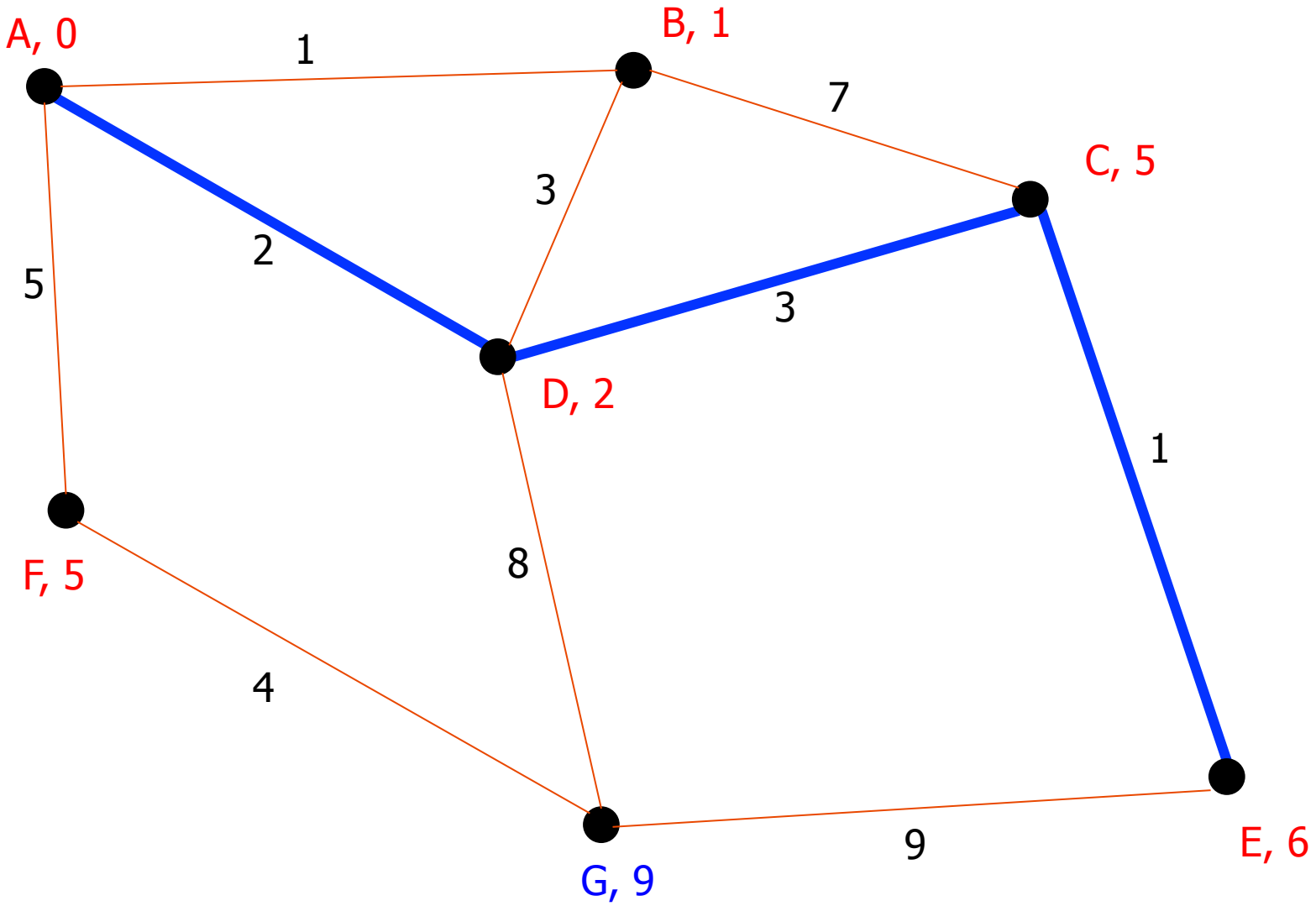
    If n.distance > current.distance + length of edge from n to current

    n.distance = current.distance + length of edge from n to current

    n.parent = current

    add n to list if it isn't there already

Planning shortest paths – Dijkstra's Algorithm



Computational Complexity of Dijkstra's algorithm

- A naive version of Dijkstra's algorithm can be implemented with a computational complexity that grows quadratically with the number of nodes.

$$\mathcal{O}(|\mathbf{V}|^{\mathbf{2}}) \tag{1}$$

- By keeping the list of nodes sorted using a clever data structure known as a priority queue the computational complexity can be reduced to something that grows more slowly

$$\mathcal{O}((|\mathbf{E}| + |\mathbf{V}|)\log(|\mathbf{V}|)) \tag{2}$$

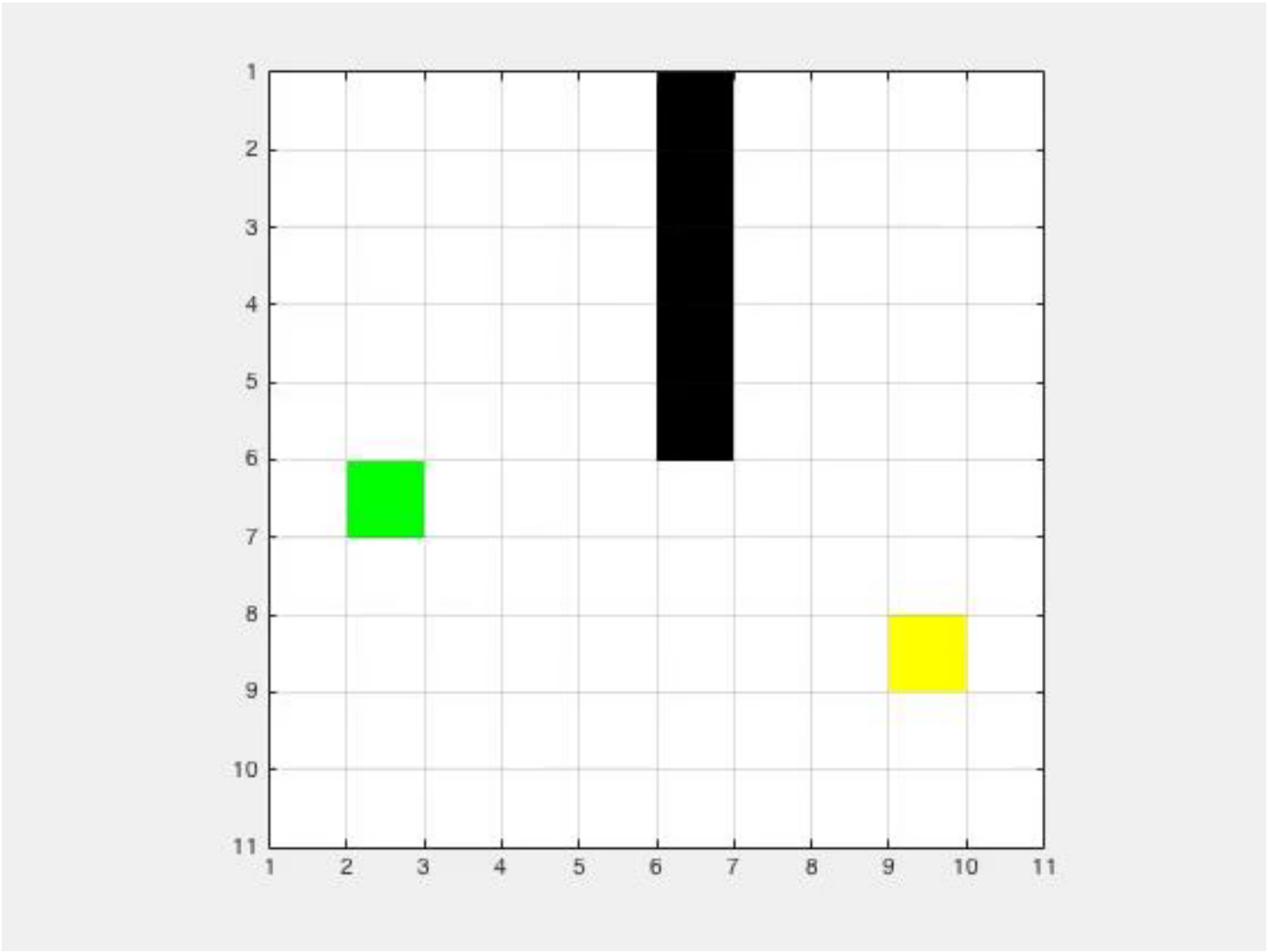
- $|\mathbf{V}|$ denotes the number of nodes in the graph and $|\mathbf{E}|$ denotes the number of edges

A-Star Procedure

SECTION 1.4

Penn

Dijkstra/Grassfire Algorithm



Dijkstra/Grassfire Algorithm

- When applied on a grid graph where all of the edges have the same length, Dijkstra's algorithm and the grassfire procedure have similar behaviors.
- They both explore nodes in order based on their distance from the starting node until they encounter the goal.

A* Search

- A* Search attempts to improve upon the performance of grassfire and Dijkstra by incorporating a heuristic function that guides the path planner.

Heuristic Functions

- Heuristic functions are used to map every node in the graph to a non-negative value
- Heuristic Function Criteria:
  - H(goal) = 0
  - For any 2 adjacent nodes x and y

    H(x) <= H(y) + d(x,y)

    d(x,y) = weight/length of edge from x to y
- These properties ensure that for all nodes, n
  - H(n) <= length of shortest path from n to goal.

Example Heuristic Functions

- For path planning on a grid the following 2 heuristic functions are often used
  - Euclidean Distance

$$H(x_n, y_n) = \sqrt{((x_n - x_g)^2 + (y_n - y_g)^2)} \tag{1}$$

  - Manhattan Distance

$$H(x_n, y_n) = |(x_n - x_g)| + |(y_n - y_g)| \tag{2}$$

  - where (x_n, y_n) denotes the coordinates of the node n and (x_g, y_g) denotes the coordinate of the goal

A* algorithm – pseudo code

- For each node n in the graph
  - n.f = Infinity, n.g = Infinity
- Create an empty list.
- start.g = 0, start.f = H(start) add start to list.
- While list not empty
  - Let current = node in the list with the smallest f value, remove current from list
  - If (current == goal node) report success
  - For each node, n that is adjacent to current

    If (n.g > (current.g + cost of edge from n to current))

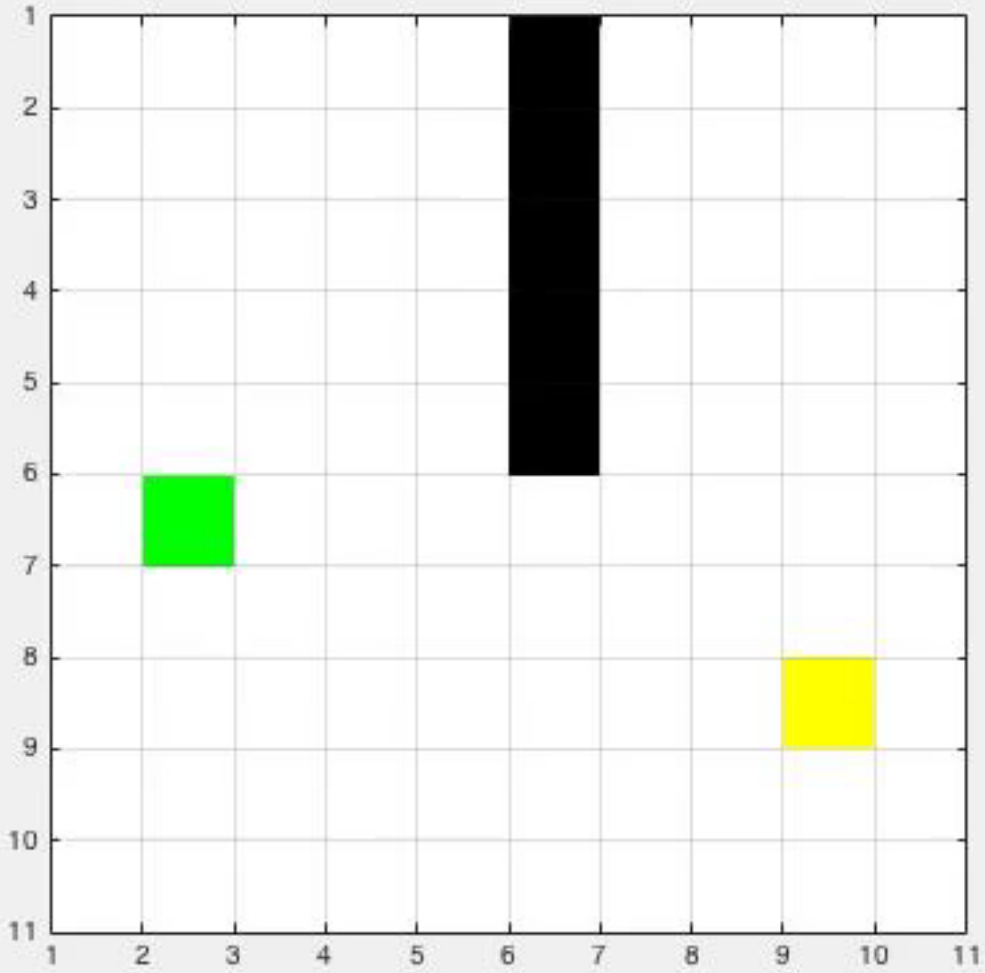
    n.g = current.g + cost of edge from n to current

    n.f = n.g + H(n)

    n.parent = current

    add n to list if it isn't there already

Dijkstra's Algorithm



A* Algorithm